SISTEMA IoT PARA MONITORIZACIÓN DEL ESTADO DE UN CENTRO DE PROCESO DE DATOS DE GRANDES DIMENSIONES

Ballesteros de Andrés, Carlos
Sypko, Denys

GRADO EN INGENIERÍA DE COMPUTADORES. FACULTAD DE INFORMÁTICA
UNIVERSIDAD COMPLUTENSE DE MADRID

Trabajo Fin de Grado en Ingeniería de Computadores

Madrid, 1 de junio de 2017

Directores:

Igual Peña, Francisco
Piñuel Moreno, Luis

Autorización de difusión

Ballesteros de Andrés, Carlos
Sypko, Denys

Madrid, a 1 de junio de 2017

Los abajo firmantes, matriculados en el Grado de Ingeniería de Computadores de la Facultad de Informática, autorizan a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Grado: "SISTEMA IoT PARA MONITORIZACIÓN DEL ESTADO DE UN CENTRO DE PROCESO DE DATOS DE GRANDES DIMENSIONES", realizado durante el curso académico 2016-2017 bajo la dirección de Francisco Igual Peña y la co-dirección de Luis Piñuel Moreno en el Departamento de Arquitectura de Computadores y Automática, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

"I have learned all kinds of things from my many mistakes. The one thing I never learn is to stop making them."

Joe Abercrombie, Last Argument of Kings

Agradecimientos

A mis padres Olena y Oleksandr por todo su apoyo y paciencia.

Denys

A mi hermana y mis padres, gracias por todo.

Carlos

A Francisco Igual Peña y a Luis Piñuel Moreno por toda la ayuda prestada.

A todos ellos, muchas gracias.

Índice general

[1]https://docs.influxdata.com/influxdb/v1.2/introduction/installation/
[2]https://docs.influxdata.com/telegraf/v1.2/introduction/installation/
[3]https://docs.grafana.org/installation/
[4]https://mosquitto.org/documentation/

Índice de figuras

Índice de códigos

Resumen

Los Centros de Proceso de Datos (CPD) son grandes instalaciones con centenares de equipos de alto rendimiento funcionando de manera concurrente. El correcto control de parámetros como la temperatura, humedad o consumo energético se convierte en un aspecto clave para un funcionamiento correcto, eficiente y seguro.

En este proyecto, se propone el despliegue de una infraestructura compuesta por un elevado número de sensores inalámbricos de bajo consumo y coste en distintos puntos de un CPD de grandes dimensiones siguiendo el paradigma IoT (*Internet of Things*), controlando y reportando periódicamente distintos parámetros como temperatura, humedad, consumo energético, presencia o humos entre otros. Dicha información será centralizada en un panel de control web que ofrezca una visión integral y en tiempo real del estado del CPD y, bajo ciertas circunstancias, tome de manera automática decisiones que aseguren el correcto funcionamiento del mismo cuando alguno de dichos parámetros supere los límites aceptables.

Además, se demuestra la flexibilidad de la solución aplicando el entorno desarrollado a dos escenarios diferentes: la monitorización de una sala fría equipada con un elevado número de servidores; y la monitorización de un laboratorio científico en el ámbito de la Física de Materiales, donde el correcto control de la temperatura resulta clave para la corrección experimental y la seguridad durante el proceso.

Palabras clave

ESP8266, NodeMCU, Adafruit Huzzah, INA219, DHT22, DS18b20, MQTT, InfluxDB, Grafana, Dashboard.

Abstract

Data Processing Centers (DPC) are large installations with hundreds of high-performance computers running concurrently. The correct control of parameters like temperature, humidity or energy consumption is a key aspect for a correct, efficient and safe operation.

In this project, we propose the deployment a high number of low-power wireless sensors at different points in a large DPC following the IoT (*Internet of Things*) paradigm, controlling and periodically reporting parameters such as temperature, humidity, energy consumption, presence or smoke among others. This information will be centralized in a web control panel that offers an integral and real time view on the state of the DPC and under certain circumstances, automatically making decisions that ensure the proper functioning of the same when any of those parameters exceeds acceptable limits.

In addition, it demonstrates the flexibility of the solution by applying the developed environment to two different scenarios: the monitoring of a cold room equipped with a high number of servers; and monitoring of a scientific laboratory in the field of materials physics, where the correct temperature control is key to the experimental correctness and security in the process.

Keywords

ESP8266, NodeMCU, Adafruit Huzzah, INA219, DHT22, DS18b20, MQTT, InfluxDB, Grafana, Dashboard.

Capítulo 1

Introducción

IoT (***Internet of Things o Internet de las Cosas***) es un paradigma que consiste en interconectar los objetos de la vida cotidiana a través de Internet. A través de este paradigma, es posible desplegar grandes redes de sensores que monitorizan, en tiempo real y de forma conjunta multitud de parámetros y pueden, en caso de ser necesario, actuar en función de los valores observados y en base a parámetros o umbrales predefinidos.

El concepto IoT se propuso en 1999 en el MIT por Kevin Ashton pero no ha sido hasta los últimos años cuando ha crecido su popularidad gracias entre otras cosas al auge de la tecnología de los smartphones. Según la empresa Gartner, en 2020 tendremos 12.863 millones unidades conectadas[1], principalmente serán los dispositivos del mercado de Domótica . Esto nos permitirá controlar diferentes aparatos integrados en nuestros hogares tales como sistemas de climatización, sistemas de seguridad y sobre todo equipos multimedia.

1.1. Objetivos y visión general del sistema

El objetivo principal de este proyecto es diseñar e implementar un sistema de bajo coste, robusto y fácilmente escalable que permita la monitorización en tiempo real de los factores

[1] http://www.gartner.com/newsroom/id/2636073

físicos que puedan tener relevancia en un Centro de Procesamiento de Datos, como humedad, temperatura, consumo o luz.

Además, resulta deseable que el sistema desarrollado pueda extenderse a otros ámbitos en los que la monitorización de cualquier proceso relevante, sin que dicha migración resulte complicada para el usuario final. Dada la versatilidad del *dashboard* de visualización utilizado, que permite seleccionar de forma muy intuitiva qué datos queremos ver y cómo hacerlo, así como programar alarmas que nos notifiquen cuando algún parámetro sobrepase los valores que le indiquemos y la multitud de sensores que se pueden conectar al nodo, se pueden encontrar otras utilidades y casos prácticos como podremos ver más adelante, véase a la hora de monitorizar el circuito de refrigeración de la facultad de Ciencias Físicas o vigilar el correcto funcionamiento de los hornos de vacío del laboratorio de Física de Materiales.

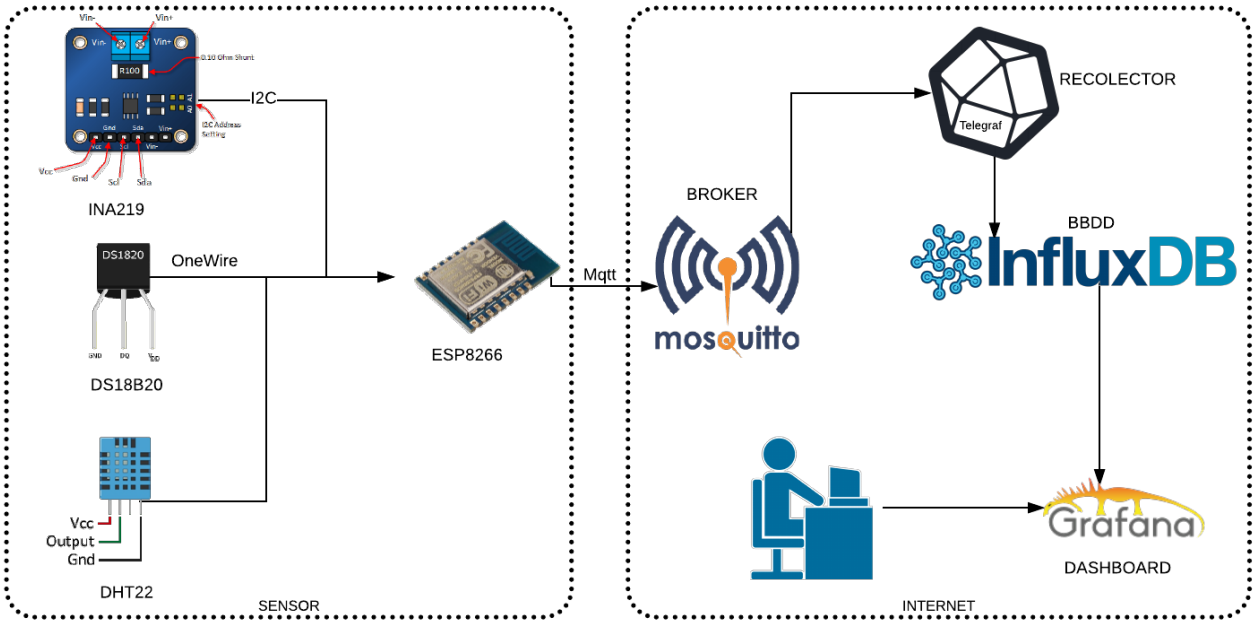


Figura 1.1: Esquema general de la infraestructura propuesta.

La Figura 1.1 ilustra un esquema general de la disposición e interrelación de los elementos que componen el sistema propuesto.

1.2. Requisitos iniciales

Para el desarrollo del sistema se pretende crear un proyecto que cumpla, la menos, con las siguientes necesidades.

1.2.1. Nodo sensor

1. Se pretende que el dispositivo tenga un **bajo coste** económico, con el fin de que sea fácilmente instalable con una inversión mínima.

2. Queremos que además conlleve un **bajo consumo**, puesto que puede ser necesario instalar un gran número de dispositivos en la misma red eléctrica e incluso que estos estén alimentados por baterías.

3. Es obvio que, puesto que puede medir parámetros críticos, ha de ser un sistema robusto, por lo que necesitamos **buen soporte a nivel software (*firmware*)**.

4. Al ser un proyecto bastante versátil, que puede ser montado en multitud de escenarios, necesitamos que el nodo sensor sea **compatible con distintos tipos e interfaces de sensor**.

5. Por último, es recomendable que el nodo que elijamos tenga un **buen soporte e nivel de comunidad** para poder resolver de forma rápida y precisa problemas con los que nos encontremos.

1.2.2. Persistencia de datos

Otra de las necesidades con la que nos encontramos es la de poder consultar de forma inmediata un histórico de los datos recogidos por los Nodos sensores, por lo que se han de

valorar diferentes sistemas de bases de datos con el fin de que este almacenaje de información se realice de la forma más eficiente posible.

1.2.3. Comunicación bidireccional

Puede darse el caso en el que necesitemos comunicarnos con el Nodo sensor de forma remota. Para ello es preciso establecer un mecanismo de recepción de mensajes en el mismo de forma que, en el supuesto de ser necesario, podamos dar órdenes o transmitir información al microcontrolador.

1.2.4. Visualización y alarmas

Obviamente todo este sistema no tendría sentido si no se pudiesen consultar los datos obtenidos de una forma fácil e intuitiva. Así que es necesario buscar algún *dashboard* o plataforma de visualización que cumpla con dichos requisitos. Además es un requisito recomendable que, de alguna forma, se nos notificara cuando los parámetros recogidos por los sensores sobrepasen unos valores críticos.

1.2.5. Seguridad

Por último, si conseguimos que la comunicación entre el microcontrolador y el *Broker* se realice de forma segura (con algún protocolo de encriptación y/o autenticación) y protegemos el acceso a los diferentes servicios podremos evitar problemas de seguridad y robos de información.

1.3. Tecnologías y recursos utilizados

Dados los anteriores requisitos, se estructura el sistema IoT desarrollado en las siguientes cuatro capas principales: dispositivo, interfaz de comunicación, persistencia y visualización,

cuya funcionalidad y características se introducen a continuación y se detallarán en el resto del documento:

- ***El dispositivo.*** Encargado de la captura de datos, en nuestro caso, un Nodo de bajo coste ESP8266 implementado en diferentes placas (NodeMCU [5] o Feather HUZZAH [7]) al que se conectan diferentes sensores como el INA219 [1], DS18B20 [3] o DHT22 [2].

- ***Interfaz de comunicación.*** Es la parte encargada de recibir y tratar la información que le envían los Nodos. Está implementado en un servidor Ubuntu al que se le han configurado servicios como Mosquitto [11] (recibe mensajes por MQTT [10]) y Telegraf [14] (trata los datos contenidos en esos mensajes).

- ***Persistencia.*** Es el eje principal del sistema para poder administrarlo de forma inteligente, esta gestionado mediante InfluxDB [13], que se encarga de almacenar los datos enviados por los Nodos.

- ***Visualización.*** Se realiza mediante el dashboard Grafana [15], que ofrece multitud de opciones de personalización para proporcionar toda la información recolectada al usuario de forma amigable, así como un sistema de alarmas.

1.4. Metodología y plan de trabajo

La finalidad de este proyecto es diseñar una infraestructura de monitorización en tiempo real basada en IoT, seleccionando, integrando y configurando herramientas ya desarrolladas, e implementando la funcionalidad requerida en el nodo sensor a través del firmware correspondiente. Para ello el trabajo se ha dividido en dos partes, una de investigación y prueba y otra de desarrollo.

Se empieza evaluando diferentes **frameworks** y lenguajes de programación para los Nodos sensores, probando hasta tres lenguajes con sus respectivos IDEs. Como se verá, se ha decidido utilizar el entorno **Arduino** [6] por su soporte a través de comunidades en Internet, la multitud de librerías desarrolladas y la familiaridad con el lenguaje de programación C, así como por la estabilidad observada al utilizarlo sobre las placas NodeMCU. Una vez decidido, se comenzaron a valorar tecnologías para configurar los servicios del servidor. Este punto es sobre todo una labor de investigación a través de foros especializados en IoT, blogs o páginas de comparativas.

Una vez decidido todo esto de forma paralela se trabaja en la programación de los Nodos sensores y en la instalación y configuración de los distintos servicios del servidor.

Con un sistema totalmente estable y funcional el trabajo ha consistido en una última fase de evaluación de diferentes formas de mejorar la infraestructura: utilización de Cloud Computing en dominios como *AWS* o *Azure*, comunicación bidireccional entre servidor y nodo sensor o añadir nuevos sensores para medir nuevos parámetros o los ya existentes.

1.5. Estructura del documento

La presente memoria se ha dividido en capítulos siguiendo la siguiente estructura:

- En el Capítulo 1 se introduce el proyecto, explicando los objetivos y requisitos de partida, las tecnologías y el plan de trabajo que se van a utilizar para llegar a los mismos.
- En el Capítulo 2 se habla de todos los aspectos relativos al Nodo Sensor. Desde la parte física y electrónica del mismo, hasta las plataformas software que se han eva-

luado y utilizado para la programación del mismo, pasando por los diferentes sensores utilizados y las interfaces de comunicación que utilizan.

- El Capítulo 3 explica del envío y recepción de los mensajes, entre el microcontrolador y el servidor de forma segura, incluyendo el tratamiento de los mismos y el almacenamiento en la base de datos.

- El Capítulo 4 trata sobre el despliegue en la nube de los distintos servicios, y de la visualización de los datos en el dashboard, así como de su sistema de alarmas. Además se comentan en esta sección casos prácticos reales donde se ha implementado el proyecto.

- En el Capítulo 5 se discuten las conclusiones alcanzadas después de desarrollar nuestro sistema, comparándolas con los requisitos iniciales y mencionando los conocimientos y habilidades que nos ha proporcionado la elaboración del mismo.

Capítulo 2

Nodo Sensor

En este capítulo explicaremos el funcionamiento del nodo, basado en el módulo **ESP8266** que es un chip de bajo costo Wi-Fi con una pila TCP/IP completa y un microcontrolador, así como de algunos de los sensores que se han instalado y configurado en el mismo. Este módulo se puede encontrar montado en diferentes placas de desarrollo. En nuestro caso hemos utilizado *NodeMCU* y *Feather Huzzah*, que además incluye un conector para baterías de tipo Lipo.

Por lo demás tanto el método para programarlas como su funcionamiento son similares, por lo que nos centraremos en temas más relevantes como la plataforma software utilizada o los sensores configurados.

2.1. Características hardware del nodo sensor

El módulo ESP8266 cuenta con un procesador que funciona a 80MHz utilizando una arquitectura de tipo RISC de 32 bits. Cuenta con una memoria RAM de 128KB y lo podemos encontrar con diferentes tamaños de memoria flash (los modelos que se han utlizado durante el desarrollo del proceso cuentan con 4MB).

En cuanto a los pines, cuenta con 16 pines GPIO, entre los que se incluye un conversor ADC de hasta 10 bits y soporte para protocolos de comunicación como I^2C, SPI o 1-wire. Podemos ver en la Figura 2.1 el pinout de un ESP8266 montado sobre un NodeMCU, del que hablamremos a continuación.

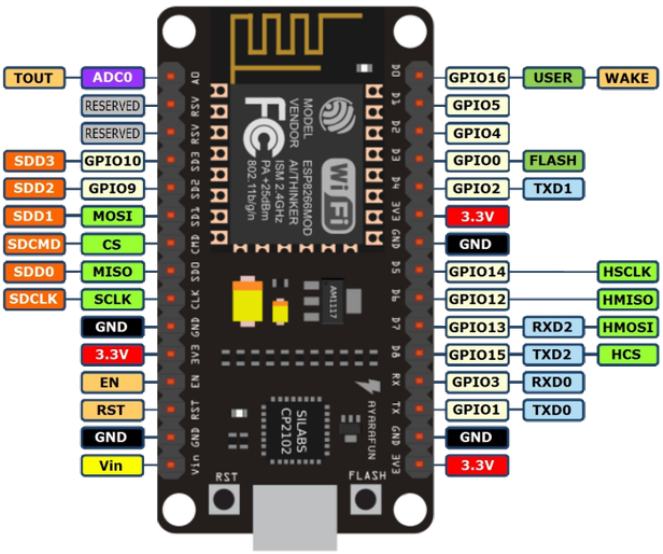


Figura 2.1: Pinout de ESP8266 sobre una placa NodeMCU.

Una de las características más interesantes de este microcontrolador, al margen de su reducido precio (en torno a 5€ comprándolo en España, menos de 3€ si recurrimos a mercados asiáticos), es que incluye un módulo de conexión inalámbrica Wi-Fi 802.11 b/g/n con toda la pila TCP/IP instalada.

Podemos encontrar el ESP8266 montado sobre placas de desarrollo de diferentes fabricantes como NodeMCU o Feather Huzzah (Figura 2.2). La diferencia entre estas placas es mínima, variando el número de pines GPIO disponibles (por ejemplo pueden sacrificarse pines para la conversión al microUSB que incluyen estas placas y que se usa para programar

el microcontrolador) o incluyendo detalles como botón de reset o de activación del Wi-Fi.

Existen diferentes alternativas a la hora de implementar sensores utilizando IoT, podemos encontrar la placa *DigiXBee*[1] que incuye soporte para redes móviles y un procesador más potente aunque es difícil encontrarla por menos de 60€, la *PADi IoT Stamp*[2] tiene unas características hardware similares a ESP8266 (quizás algo inferiores en cuanto a memoria, pero cuenta con un procesador ligeramente más potente) y su precio es de apenas 2 dólares, sin embargo la comunidad de desarrolladores es mucho menor. Otra alternativa es la *Pycom WiPy*[3] con un precio moderado (unos 20€) y unas características hardware muy interesantes (32MB de Flash y procesador de doble núcleo) aunque sólo permite programarla con *MicroPython* y tiene muchas menos librerías desarrolladas.

Figura 2.2: Feather Huzzah vs ESP8266 vs Node MCU.

2.2. Infraestructuras software evaluadas

Para el desarrollo del proyecto se han testado diferentes plataformas de programación con sus correspondientes entornos (podemos ver alguno de ellos en la Figura 2.3) con el fin de seleccionar la que más se adapta a nuestras necesidades. Para ello se realizaron pruebas

[1] https://www.digi.com/lp/xbee/hardware
[2] https://www.pine64.org/?page_id=917
[3] https://www.pycom.io/product/wipy/

básicas de conexión **MQTT** y lectura de datos a través de un sensor con las tres herramientas seleccionadas:

- ***MicroPython.*** De acuerdo con la definición del sitio oficial, MicroPython es un pequeño pero eficiente interprete del Lenguaje de Programación Python 3 que incluye un subconjunto mínimo de librerías y que además está optimizado para que pueda correr en microcontroladores e IoT. Permite programar a más alto nivel que LUA y Arduino, y a pesar de evaluarse se desechó del proyecto porque gran parte de las librerías necesarias para la implementación del mismo aún no habían sido implementadas.

- ***LUA.*** Es un lenguaje desarrollado en la Pontificia Universidad Católica de Río de Janeiro. Al ser tan ligero y compacto se está usando en la actualidad para programar controladores, hardware e IoT. A pesar de la gran comunidad que tiene detrás en internet, y de que se llegó a programar una primera versión del proyecto en este lenguaje, encontramos problemas a la hora de configurar el modo DeepSleep, que permite "dormir" el dispositivo entre lecturas de datos y envíos MQTT, por lo que también decidimos desechar este lenguaje.

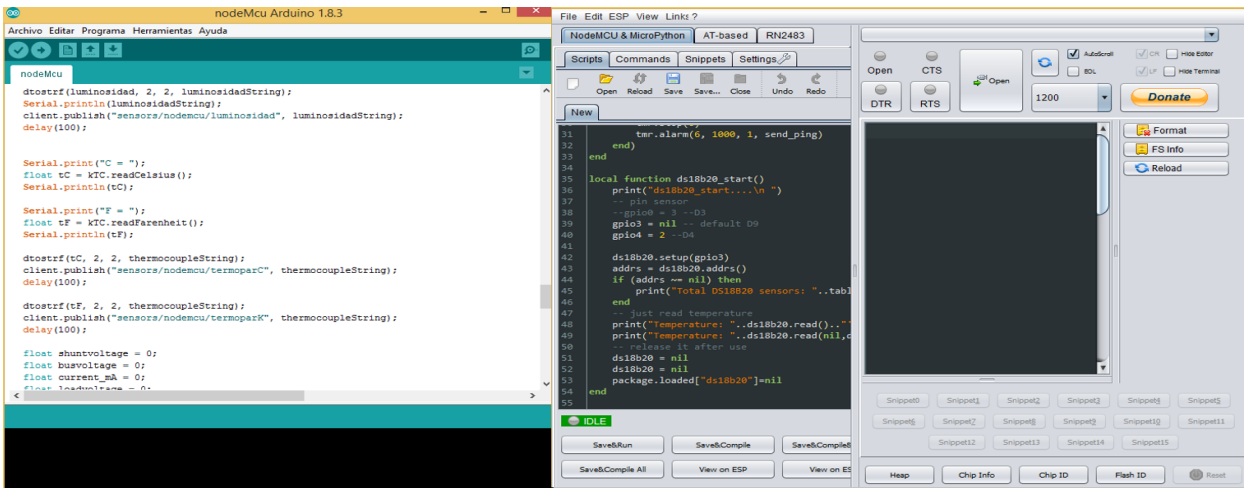


Figura 2.3: ArduinoIDE vs ESPlorerIDE (*MicroPython y LUA*).

- ***Arduino.*** A mediados del año 2016 una comunidad de desarrolladores portaron el ESP8266 al IDE de Arduino, lo que permitió que el NodeMCU y el Feather Huzzah pudieran ser programados desde esta plataforma. El IDE de Arduino es un framework muy robusto y estable que además cuenta con un gran número de librerías, sobre todo de sensores pero también de protocolos de seguridad como TLS, lo que añadido a la familiaridad con el lenguaje de programación (muy similar a C) nos llevó a decantarnos por él.

2.3. Interfaces y protocolos de comunicación

El controlador **ESP8266** soporta diversos protocolos para la comunicación con periféricos. Además una de las ventajas que nos ofrecía el IDE de Arduino es que las empresas propietarias de los diferentes protocolos proporcionan librerías para que sean fácilmente utilizables en esta plataforma.

Las interfaces de comunicación que hemos utilizado para la conexión lógica entre los sensores y la placa son los siguientes:

- ***One Wire.*** Es un protocolo de comunicaciones en serie diseñado por la empresa Dallas Semiconductor. Está basado en un bus que comparten el maestro y varios esclavos y que sólo tiene una línea de datos, que se usa también para la alimentación.
  La comunicación es sencilla: los periféricos se identifican de forma unívoca mediante direcciones de 8 bytes, y pulsos eléctricos a través del bus de una determinada duración representan el 0 o el 1 lógicos.

- I^2C**.** Este protocolo fue diseñado en 1982 por Philips. Está pensado para funcionar en un bus compartido utilizando la arquitectura *Maestro-esclavo* y de forma síncrona.
  La transferencia de datos, cuyo esquema podemos ver en la Figura 2.4, es siempre

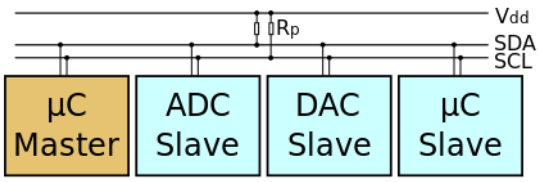


Figura 2.4: Esquema de funcionamiento de un bus I^2C con un maestro y varios esclavos.

inicializada por un maestro y el esclavo reacciona. Es posible tener varios maestros mediante un modo multimaestro, en el que se pueden comunicar entre si, de modo que uno de ellos trabaja como esclavo.

- ***SPI.*** El protocolo SPI (Serial Peripheral Interface) es un estándar para controlar periféricos digitales que utilicen comunicación síncrona en serie (bit a bit).
  Incluye varias líneas, una de reloj, dato entrante, dato saliente y un pin de chip select, que conecta o desconecta la operación del dispositivo con el que uno desea comunicarse. De esta forma, este estándar permite multiplexar las líneas de reloj. En la Figura 2.5, que podemos ver a continuación, se muestra un ejemplo básico de la conexión mediando SPI entre un maestro y tres esclavos.

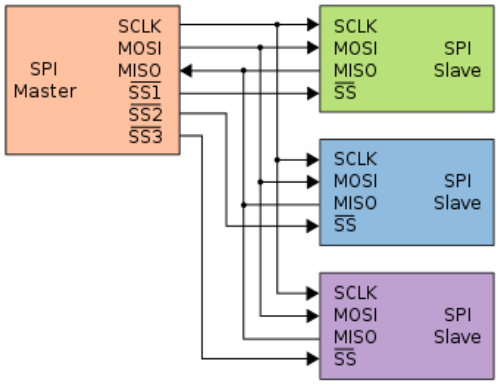


Figura 2.5: Esquema de funcionamiento de un bus SPI con un maestro y varios esclavos.

2.4. Obtención de datos

Hasta ahora hemos visto el controlador ESP8266, las placas de desarrollo que lo montan y las infraestructuras software utilizadas para programarlo, a continuación hablaremos de algunos de los sensores usados para la obtención de los datos, y que utilizan diferentes interfaces para comunicarse con el ESP8266.

Se han probado varios diferentes, como pueden ser el **INA219** (protocolo I^2C) para medir parámetros de corriente continua, **DS18B20** o **DHT22** (protocolo OneWire) que captan temperatura y temperatura y humedad respectivamente, una Fotoresistencia (**LDR**) conectada a un conversor analógico digital o un Termopar **MAX31855K**[4]. Los dos últimos realizan el envio de los datos a través de una interfaz SPI, pero únicamente se explicará a continuación un sensor por interfaz de comunicación o protocolo.

2.4.1. Sensor de temperatura

Se han utilizado en el proyecto sensores DS18B20, fabricados por la empresa Dallas Semiconductor. Se trata de un termómetro digital con una precisión de $\pm 0.5^{o}$C y un precio muy económico. Su instalación, tal como podemos ver en la Figura 2.6, es bastante sencilla puesto que sólo se necesita de una resistencia de 4.7 KOhm.

Este sensor se comunica con la placa mediante el protocolo **OneWire**, y puede encontrarse en el mercado en diferentes formatos. En el proyecto se han probado la versión optimizada para placas de prototipado y una sumergible.

A continuación se muestra el código necesario para inicializar el bus **OneWire** y leer la temperatura utilizando este sensor en los lenguajes Arduino (Código 2.1), LUA(Código 2.2)

y MicroPython(Código 2.3).

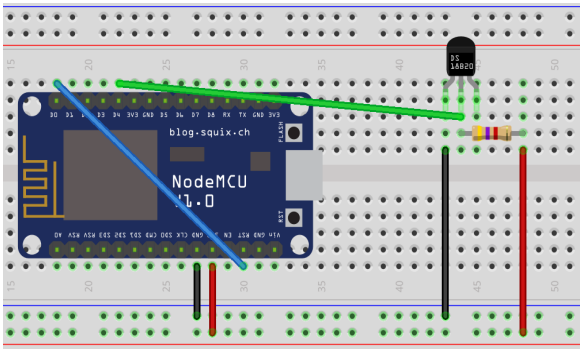


Figura 2.6: Esquema de conexión del sensor DS18B20.

```
#include <OneWire.h>
#include <DallasTemperature.h>
#define DS18B20_PIN D4 //Pin declarado para la comunicación OneWire
...
OneWire oneWire(DS18B20_PIN);
DallasTemperature DS18B20(&oneWire);
DS18B20.begin(); //Se inicia el sensor
...
float getTemperature() {
  float temp;
  do {
    DS18B20.requestTemperatures();
    temp = DS18B20.getTempCByIndex(0);
    delay(100);
  } while (temp == 85.0 || temp == (-127.0));
  return temp;
}
```

Código 2.1: Arduino. Sensor DS18B20.

```
local function ds18b20_start()
  --Pin del sensor
  gpio4 = 2 --D4
  ds18b20.setup(gpio4)
  --Leamos la temperatura
  print("Temperature:    "..ds18b20.read().." 'C\n")
  print("Temperature: "..ds18b20.read(nil,ds18b20.K).." 'K")
end
```

Código 2.2: Lua. Sensor DS18B20.

```
import time
import machine
import onewire
#Pin del sensor GPIO2/D4
dat = machine.Pin(2)
#Creamos el objeto OneWire
ds = onewire.DS18B20(onewire.OneWire(dat))

#Escaneamos los dispositivos del bus
roms = ds.scan()
print('found devices:', roms)

ds.convert_temp()
time.sleep_ms(750)
for rom in roms:
  print(ds.read_temp(rom))
```

Código 2.3: MicroPython. Sensor DS18B20.

2.4.2. INA219

El INA219 es un chip fabricado por Texas Instruments (aunque pueden encontrarse clones en los mercados asiáticos), que permite medir parámetros de corriente continua como son voltaje e intensidad de entrada con una precisión de un 1 %, lo cual nos permite calcular de forma trivial otros parámetros relacionados como son la resistencia eléctrica o el consumo de potencia en vatios. El esquema de conexión con la placa Feather Huzzah es el mostrado en la Figura 2.7.

El código para inicializar el bus I^2C y realizar las lecturas del sensor y los cálculos necesarios corresponde con el Código 2.4.

```
#include <Adafruit_INA219.h>
#include <Wire.h>
...
Adafruit_INA219 ina219; //I2C
...
//Se inicia el sensor
Wire.begin(4,5); //I2C -> sda, scl
ina219.begin();
...
```

```
//Lectura de los datos
float shuntvoltage = ina219.getShuntVoltage_mV();
float busvoltage = ina219.getBusVoltage_V();
float current_mA = ina219.getCurrent_mA();
float loadvoltage = busvoltage + (shuntvoltage / 1000);
float power_mW = (current_mA) * loadvoltage;
```

Código 2.4: Arduino. Sensor INA219.

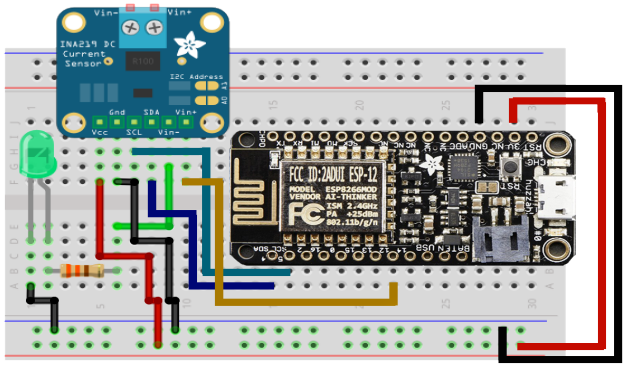


Figura 2.7: Esquema de conexión del sensor INA219.

2.4.3. Fotoresistencia

El sensor de luz se ha configurado mediante una fotorresistencia conectada a un conversor analógico digital MCP3008 (Figura 2.8), que se comunica con el controlador mediante el protocolo **SPI**.

El LDR (Light Dependent Resistor) o resistencia dependiente de la luz o también fotocélula, es una resistencia que varía su resistencia en función de la luz que incide sobre su superficie. Cuanto mayor sea la intensidad de la luz que incide en la superficie del LDR menor será su resistencia y cuanto menos luz incida mayor será su resistencia y por tanto menor es la lectura de voltaje que se realiza en el ESP8266.

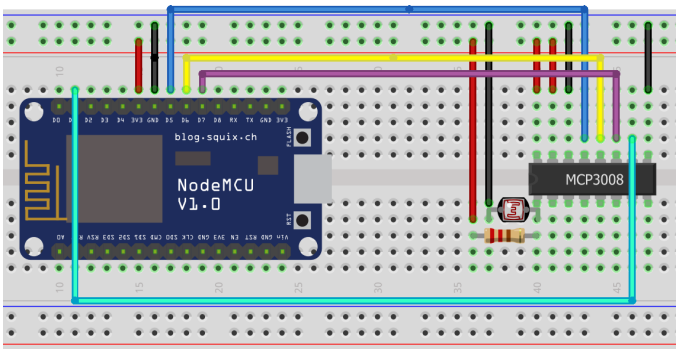


Figura 2.8: Esquema de conexión del sensor LDR con MCP3008.

A continuación, en el Código 2.5 se detallan las instrucciones necesarias para la inicialización, configuración y lectura de los parámetros proporcionados por este sensor:

```
#include <SPI.h>
#include <MCP3008.h> //Conversor ADC
//** Definiciones **//
#define CS_PIN 14
#define CLOCK_PIN 5
#define MOSI_PIN 13
#define MISO_PIN 12
...
//Librería MCP
MCP3008 adc(CLOCK_PIN, MOSI_PIN, MISO_PIN, CS_PIN);
...

//Lectura de los datos
int val = adc.readADC(0);   // leamos Canal 0 de MCP3008 ADC(pin 1)
float voltage = (val * 3.3) / 1023 ; //Conversión de adc con voltaje 3.3
```

Código 2.5: Arduino. Sensor LDR y MCP3008.

2.5. Evaluación de autonomía

Con el fin de que el sistema pueda instalarse sin dependencia de una línea de corriente eléctrica o bien de tener un respaldo en caso de corte del suministro eléctrico y puesto que la placa Feather Huzzah incluye una entrada de alimentación de tipo Lipo se han probado diferentes baterías para comprobar la duración de la vida de las mismas y por tanto que uso

podrían tener.

El Primero de los modelos probados es del fabricante PKCELL, más concretamente el modelo **LP503035**(Figura 2.9), con una capacidad de 500mAh y un voltaje de salida de 3,7V. Esta batería puede encontrarse en mercados asiáticos por un precio que oscila los 2,5$.

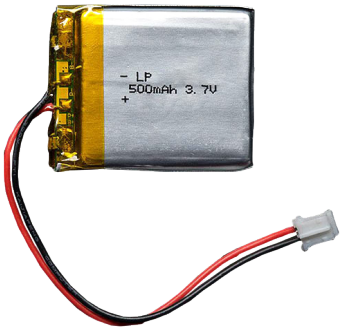


Figura 2.9: Primera de las baterías probadas para el proyecto.

Este modelo dio unos resultados de autonomía relativamente buenos, utilizando en el microcontrolador el modo **DeepSleep** y haciendo que éste despertara cada sesenta segundos, se consiguió una duración de la vida útil de la batería algo superior a cincuenta horas. Por lo que se planteó que esta pila podría utilizarse para instalar el Nodo sensor de modo que que sea independiente de la red eléctrica.

El segundo de los tipos de batería probados es una pila de botón de Litio recargable, conectada aun adaptador que permite la conexión de tipo Lipo, como puede verse en la Figura 2.10.

Estas baterías tienen una capacidad de 120mAh, sin embargo su voltaje de salida es de

3,6V, lo que hace que se encuentre «en el límite» por lo que la duración de su vida útil bajo las mismas condiciones apenas llega a ocho horas. Sin embargo su bajo coste hace que deba ser tenida en cuenta como pila de respaldo para hipotéticos cortes de luz en caso de que el nodo sensor estuviese conectado a la red eléctrica.

Figura 2.10: Batería conectada a adaptador Lipo.

Capítulo 3

Intercambio y almacenamiento de datos

En este capítulo veremos como se realiza la comunicación de los datos, desde que los sensores de la ESP8266 los capturan hasta su llegada a la Base de Datos, más detalladamente hablaremos del protocolo de comunicación **MQTT**, del **Broker** usado y de **Telegraf** que se encarga de recolectar todos los datos, procesarlos si fuera necesario y mandarlos a **InfluxDB**.

3.1. MQTT

MQTT es un protocolo usado para la comunicación **machine-to-machine**(M2M) en el *Internet of Things*. Es un protocolo cliente-servidor sobre el que se publican/suscriben los mensajes entre los dispositivos interconectados. Está orientado principalmente a la comunicación de sensores, debido a su bajo consumo de banda ancha y mínima sobrecarga puede ser utilizado en dispositivos con pocos recursos. Los mensajes se trasmiten por TCP-IP. Puede verse en el Código 3.1 la configuración para la inicialización del servicio y el envío de mensajes en nuestro poyecto.

```
#include <PubSubClient.h>
//Función para recibir los mensajes MQTT
void mqtt_sus (char* topic, byte* payload, unsigned int length);
//Parámetros de conexión
const char* mqtt_server = "147.96.67.172";
const char* mqtt_user = "tfg-esp";
const char* mqtt_pass = "*********";
```

```
//TLS utiliza el puerto 8083
WiFiClientSecure wifiClient;
//Inicializamos la comunicación
PubSubClient client(mqtt_server, 8883, wifiClient);
client.connect(clientId.c_str(), mqtt_user, mqtt_pass);
//Envío de un mensaje con un determinado topic
client.publish("sensors/nodemcu/temperatureDht", temeperatureDhtString);
```

Código 3.1: Arduino. Configuración MQTT.

La arquitectura de MQTT sigue una topología de estrella, con un programa o dispositivo que actúa como servidor (*Broker*) el cual es el encargado de gestionar la red e intercambiar los mensajes. La comunicación se basa en los **topics**, que el cliente que publica el mensaje crea y a los que los nodos que pretenden consumirlo se suscriben. Un **topic** se representa mediante una cadena y tiene una estructura jerárquica. Cada nivel de la jerarquía se separa con el símbolo (/). De este modo, un posible ejemplo de jerarquía desarrollado en el ámbito de nuestro proyecto podría ser **edificio1/planta1/sala1/arduino0/temperatura**; la Figura 3.1 muestra la jerarquía MQTT planteada en el ámbito de nuestro sistema de monitorización de un centro de proceso de datos.

Otra de las ventajas destacadas de este protocolo es que ofrece tres calidades de servicio para la entrega de mensajes:

- **Como máximo una vez**, a lo sumo el mensaje publicado se recibe una vez. Se puede producir pérdida de mensajes.
- **Al menos una vez**, donde se asegura que los mensajes llegan, pero se pueden producir duplicados.
- **Exactamente una vez**, se asegura que los mensajes llegan exactamente una sola vez.

Se consideraron otros protocolos de comunicación, como HTTP, pero por la naturaleza de los dispositivos IoT, donde suele ser importante un bajo consumo de recursos, lo habitual

es utilizar protocolos optimizados como son MQTT. Al fin y al cabo todas estas características lo hacen ideal en este tipo de ecosistemas.

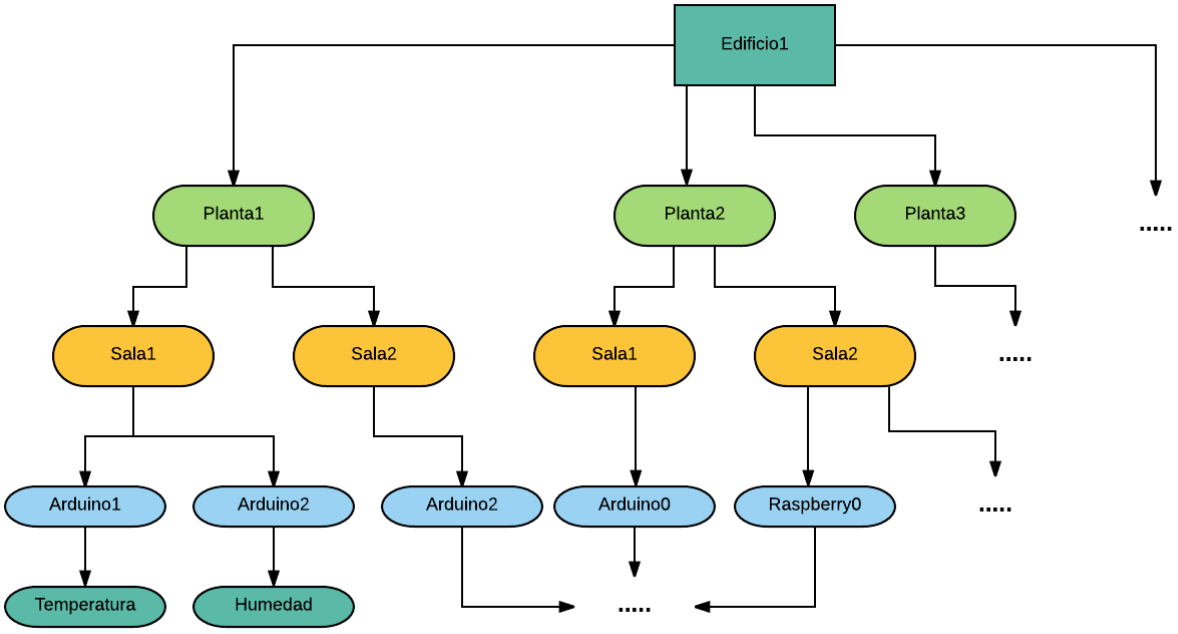


Figura 3.1: Jerarquía MQTT planteada.

3.1.1. Broker

Es precisamente el Broker el elemento encargado de gestionar la red y redirigir los mensajes a sus correspondientes suscriptores. En nuestro caso, se optó por utilizar uno de los Brokers más conocidos que existen para MQTT: **Mosquitto**. Mosquitto es un Broker desarrollado como código abierto por Eclipse[1], ampliamente utilizado debido a su ligereza frente a otras alternativas como **Mosca** que al estar escrito en java consume más recursos, lo que nos permite fácilmente emplearlo en gran número de ambientes, incluso si éstos pueden

[1] http://mosquitto.org/

aportar pocos recursos. A través de esta página[2] se puede descargar y consultar la documentación necesaria para instalarlo y usarlo en diferentes sistemas operativos.

Dentro de la estructura que aparece en la Figura 3.1, nuestros únicos **«emisores»** son los sensores de Temperatura y Humedad que hemos colocado en la Planta1->Sala1->Arduino1 y Arduino2. Cada uno de ellos, lo vamos a asignar a un topic propio quedando el listado de topics de la siguiente forma:

Sala1:

- edificio1/planta1/sala1/arduino0/temperatura
- edificio1/planta1/sala1/arduino0/humedad

Para poder ver los mensajes publicados por los emisores hay que subscribirse a los dos topics anteriores.

El Broker es una de las piezas principales del sistema ya que sin **MQTT** no tendríamos una forma tan eficiente de comunicación. Cabe constatar la sencillez con la que se puede instalar el sistema y tenerlo funcionando de forma instantánea. En nuestro proyecto el *Broker* está configurado para escuchar los mensajes a través de los puertos 1883 (por defecto) y 8883 (seguro).

3.2. Telegraf

Es un agente, escrito en *Go*, que se encarga de recopilar métricas y distribuirlas a una alta gama de salidas, como pueden ser dashboards o bases de datos, de forma nativa. Este sistema se extiende no sólo a las salidas, sino también a las posibles entradas, que pueden ser

[2] https://mosquitto.org/download/

las características del propio servidor como CPU/RAM/LOAD, ping a máquinas o también servicios en MQTT, StatsD o TCP. Principalmente ofrece los siguientes servicios:

- **Recogida de los datos,** de diferentes fuentes.
- **Procesado de los datos,** para transformar y formatear los mismos.
- **Estadística de los datos** como media, mínimo, máximo, cuantiles, etc.
- **Salida de los datos,** para redistribuir los mismos a distintas aplicaciones.

En nuestro caso sólo utilizamos Telegraf para leer los datos desde Mosquitto y escribirlos en InfluxDB, puesto que el procesado de datos para visualización se realiza desde Grafana. A continuación se muestra un posible ejemplo de como se modifican las entradas y salidas en el fichero de configuración (/etc/telegraf/telegraf.conf) :

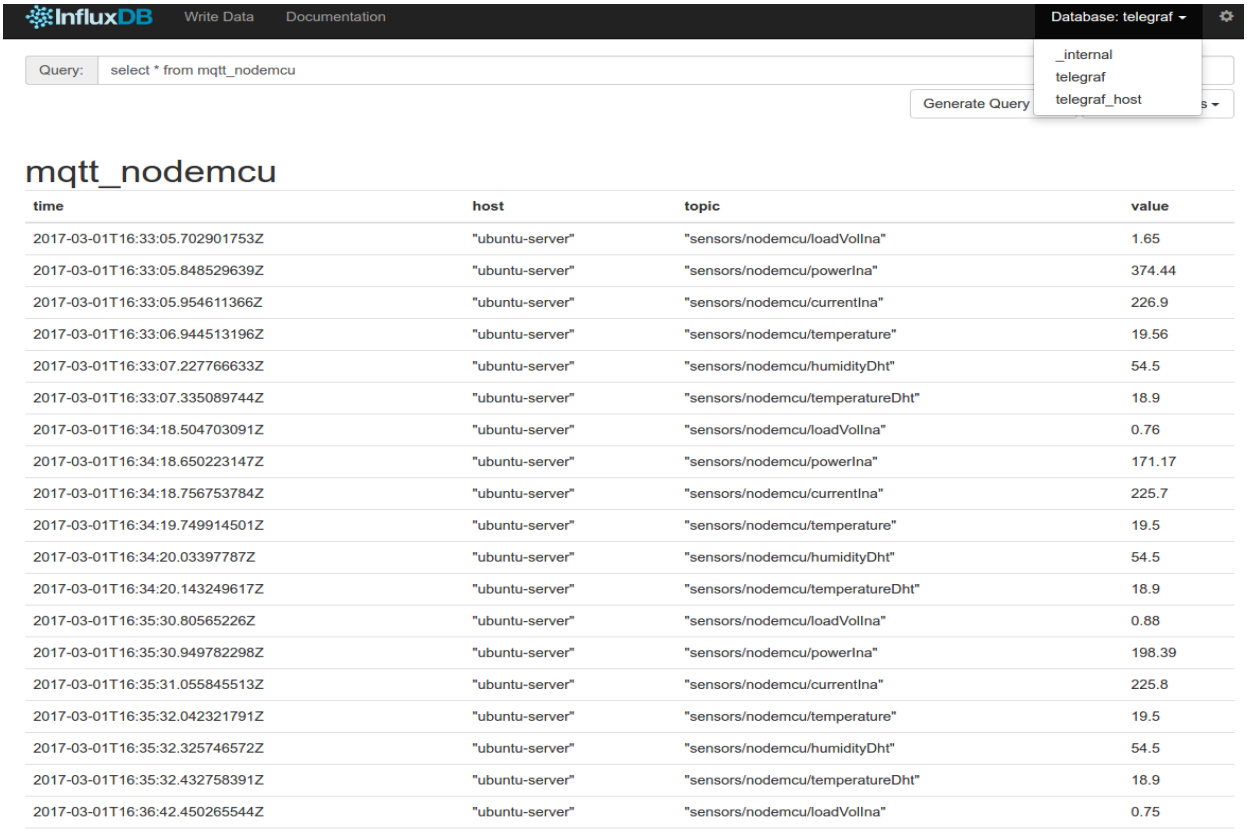
```
# OUTPUTS
[[outputs.influxdb]]
  url = "http://192.168.59.103:8086" # required.
  database = "telegraf" # required.
  precision = "s"
  ...

# INPUTS
[[inputs.cpu]]
  percpu = true
  totalcpu = false
  # filter all fields beginning with 'time_'
  fielddrop = ["time_*"]
  ...
```

Hay aplicaciones similares a Telegraf pero el hecho de que esté programada en GO al igual que InfluxDB, implica una buena integración de ambas, lo cual nos hizo decidirnos por ella por delante de otras alternativas como por ejemplo Node-RED.

3.3. Persistencia de datos

Como hemos ya comentado anteriormente para almacenamiento de datos hemos utilizado la base de datos no relacional **InfluxDB**. Es un proyecto Open Source gestionado por InfluxData, también escrito en GO y optimizado para almacenamiento de series de datos temporales, logrando una gran eficiencia tanto en espacio de almacenamiento como en lectura y escritura información.



mqtt_nodemcu

time	host	topic	value
2017-03-01T16:33:05.702901753Z	"ubuntu-server"	"sensors/nodemcu/loadVolIna"	1.65
2017-03-01T16:33:05.848529639Z	"ubuntu-server"	"sensors/nodemcu/powerIna"	374.44
2017-03-01T16:33:05.954611366Z	"ubuntu-server"	"sensors/nodemcu/currentIna"	226.9
2017-03-01T16:33:06.944513196Z	"ubuntu-server"	"sensors/nodemcu/temperature"	19.56
2017-03-01T16:33:07.227766633Z	"ubuntu-server"	"sensors/nodemcu/humidityDht"	54.5
2017-03-01T16:33:07.335089744Z	"ubuntu-server"	"sensors/nodemcu/temperatureDht"	18.9
2017-03-01T16:34:18.504703091Z	"ubuntu-server"	"sensors/nodemcu/loadVolIna"	0.76
2017-03-01T16:34:18.650223147Z	"ubuntu-server"	"sensors/nodemcu/powerIna"	171.17
2017-03-01T16:34:18.756753784Z	"ubuntu-server"	"sensors/nodemcu/currentIna"	225.7
2017-03-01T16:34:19.749914501Z	"ubuntu-server"	"sensors/nodemcu/temperature"	19.5
2017-03-01T16:34:20.03397787Z	"ubuntu-server"	"sensors/nodemcu/humidityDht"	54.5
2017-03-01T16:34:20.143249617Z	"ubuntu-server"	"sensors/nodemcu/temperatureDht"	18.9
2017-03-01T16:35:30.80565226Z	"ubuntu-server"	"sensors/nodemcu/loadVolIna"	0.88
2017-03-01T16:35:30.949782298Z	"ubuntu-server"	"sensors/nodemcu/powerIna"	198.39
2017-03-01T16:35:31.055845513Z	"ubuntu-server"	"sensors/nodemcu/currentIna"	225.8
2017-03-01T16:35:32.042321791Z	"ubuntu-server"	"sensors/nodemcu/temperature"	19.5
2017-03-01T16:35:32.325746572Z	"ubuntu-server"	"sensors/nodemcu/humidityDht"	54.5
2017-03-01T16:35:32.432758391Z	"ubuntu-server"	"sensors/nodemcu/temperatureDht"	18.9
2017-03-01T16:36:42.450265544Z	"ubuntu-server"	"sensors/nodemcu/loadVolIna"	0.75

Figura 3.2: Información en InfluxDB.

Soporta cientos de miles de escrituras por segundo, además de clustering. Los datos pueden ser etiquetados, lo que junto con un lenguaje de consultas similar a SQL facilita búsquedas flexibles. Dispone de una API REST, ademas de multitud de clientes para dife-

rentes lenguajes.

Los principales motivos por los cuales hemos elegido esta base de datos por delante de otras alternativas también evaluadas como MongoDB o CouchDB son su fácil integración con nuestro ecosistema de recolección de métricas y la herramienta de visualización así como los resultados que pudimos ver después de compararlos en sitios como DB-Engines[3], página que compara infraestructuras en función de diferentes parámetros. La información se guarda como measurement *mqtt_ ** y con sus valores de *host*, *topic*, *value*, tal como lo refleja la Figura 3.2.

3.4. Seguridad

La seguridad, es un aspecto muy relevante en paradigma IoT, se puede definir como aquellas actividades enfocadas a proteger un determinado dispositivo, servicio o la comunicación entre ellos. Los principales riesgos son robo de información y control y uso malintencionado de los dispositivos.

La primera medida que se utilizó fue la de requerir **autenticación** a la hora de acceder a todos los servicios (InfluxDB, Mosquitto, Grafana y Telegraf) con diferentes contraseñas para que, en el supuesto de que se descifrara la de uno de ellos, el resto permanecieran a salvo.

No obstante al poco tiempo y haciendo pruebas se descubrió que simplemente con el uso de un sniffer, como puede ser Wireshark, se podría (capturando el paquete adecuado) ver la información del mensaje MQTT, incluyendo la contraseña del Mosquitto (Figura 3.3), por lo que se decidió cifrar la comunicación entre Broker y Nodo.

[3] https://db-engines.com/en/

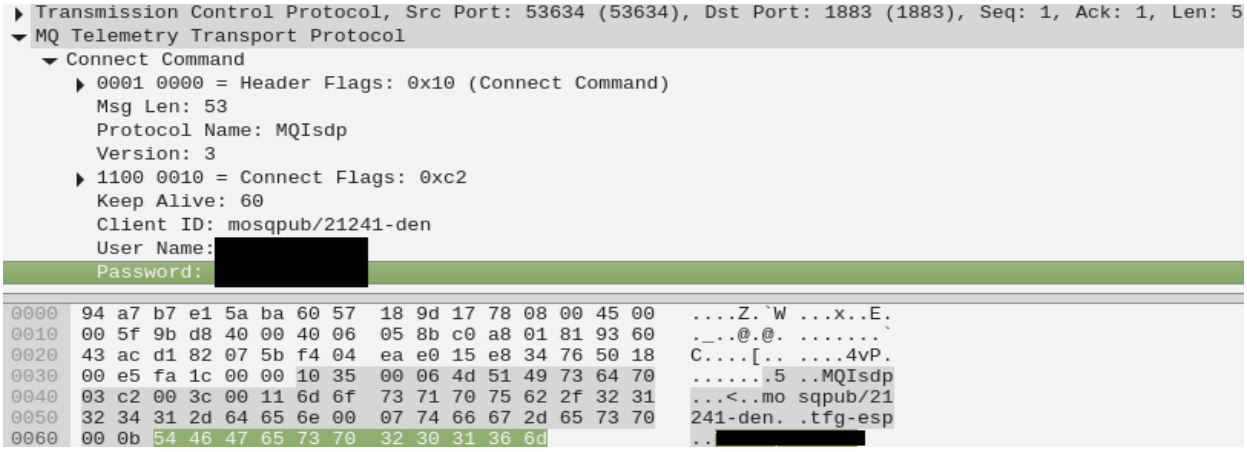


Figura 3.3: Captura de un paquete en el que se aprecia el login y password.

Es por ello que llegados a este punto se decidió proteger la comunicación con algún sistema de encriptación, optándose por el estándar **TLS** [12] *(Transport Layer Security)*, que se trata de un protocolo ubicado en la capa de transporte basado en criptografía asimétrica o de clave pública.

Lo primero que se realizó fue la creación de los certificados en el servidor, para lo que tuvimos que instalar en el mismo *OpenSSL* y ejecutar este **script**[4], que nos generó un certificado autofirmado así como la clave privada del servidor.

A continuación configuramos Mosquitto para que funcionase con estos certificados, añadiéndolos en el directorio de instalación del servicio e indicando, desde el fichero de configuración del mismo, el puerto de escucha con este protocolo y la ubicación de los ficheros, como se muestra en el siguiente ejemplo:

[4] https://github.com/owntracks/tools/blob/master/TLS/generate-CA.sh

```
# MQTT over TLS/SSL

listener 8883
# Certificado de Autoridad público
cafile /etc/mosquitto/certs/ca.crt
# Certificado público del servidor donde corre el broker
certfile /etc/mosquitto/certs/hostname.crt
# Clave privada del servidor
keyfile /etc/mosquitto/certs/hostname.key
```

El siguiente paso fue configurar los Nodos para que enviasen los mensajes MQTT cifrados al Broker. Para realizar este paso hay que subir el *Certificado de Autoridad público* mencionados anteriormente a la memoria flash del ESP8266, utilizando un plugin del Arduino IDE.

Por último hubo que modificar el código del programa, tal como se muestra en el Código 3.2, leyendo el fichero al principio del programa, y enviando los mensajes de forma segura.

```
//Librería necesaria
#include <WiFiClientSecure.h>
...
//Inicialización de la conexión segura;
WiFiClientSecure wifiClient;
PubSubClient client(mqtt_server, 8883, wifiClient);
...
//Lectura del fichero
if (!SPIFFS.begin()) {
  Serial.println("Failed to mount file system");
  return;
}
File ca = SPIFFS.open("/ca.crt", "r");
Serial.println(ca.size());
if (!ca)
  Serial.println("Error to open ca file");
else
  Serial.println("Success to open ca file");
if(wifiClient.loadCertificate(ca))
  Serial.println("loaded");
else
```

```
Serial.println("not loaded");
```

Código 3.2: Arduino. Lectura del certificado.

Una vez seguidos todos estos pasos, cuando se repitió el proceso de capturar paquetes MQTT utilizando Wireshark se puede comprobar en la Figura 3.4 que ya no se podía consultar la información contenida en los mensajes.

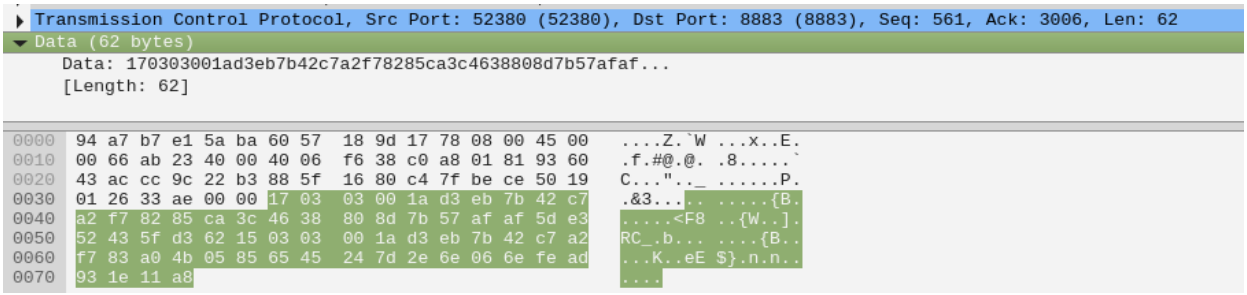


Figura 3.4: Captura de un paquete encriptado mediante TLS, se aprecia que no se muestra nada de información.

3.5. Comunicación bidireccional

Otra de las funcionalidades añadidas al Nodo sensor fue la posibilidad de que el Nodo reciba mensajes MQTT. Para ello se debe de configurar a que topic se tiene que suscribir al arrancar la conexión, así como a que función ha de llamarse cuando se reciba un mensaje con dicho topic. En nuestro caso se probó encendiendo y apagando leds, conectados al microcontrolador, de forma remota.

En el Código 3.3 se muestra la incialización y los métodos necesarios para poner en marcha esta funcionalidad:

```
//** Definiciones **//
void mqtt_sus (char* topic, byte* payload, unsigned int length);
...

WiFiClientSecure wifiClient;
```

```
PubSubClient client(mqtt_server, 8883, wifiClient);

void setup() {
   ...
   client.setCallback(mqtt_sus);
}

void reconnect() {
  while (!client.connected()) {
    String clientId = "ESP8266Client-";
    clientId += String(random(0xffff), HEX);

    if (client.connect(clientId.c_str(), mqtt_user, mqtt_pass)) {
      Serial.println("connected");
      client.subscribe("ledStatus"); //Nodo se suscribe al topic "ledStatus"
    }
    else {
      Serial.print("failed, rc=");
      Serial.print(client.state());
      Serial.println(" try again in 5 seconds");
      delay(2000);
    }
  }
}

void loop()
{
 if (!client.connected()) {
  reconnect();
 }
 ...
 client.loop(); //Permite al MqttCliente procesar los mensajes entrantes
}
```

Código 3.3: Arduino. Comunicación Bidireccional.

La aplicación práctica de esta funcionalidad radica en la posibilidad de configuración de ciertos parámetros del nodo sensor desde el prisma del usuario (por ejemplo, desde el dashboard); por ejemplo, parámetros como frecuencia de muestreo, dimensiones concretas a muestrear, o unidades de medida a utilizar podrían configurarse remotamente y de forma individual para cada nodo sensor.

Capítulo 4

Visualización de los datos, alarmas y despliegue del proyecto

Durante la fase de programación de los ESP8266, cuando aún no se tenía definida la arquitectura de aplicaciones en el servidor, se utilizó la plataforma **ThingsPeak** [16](Figura 4.1). Se trata de una aplicación web, basada en el envío de mensajes mediante el protocolo HTTP, la cual nos sirvió para comprobar el correcto funcionamiento de los periféricos conectados a la placa. Es bastante intuitiva y fácil de utilizar, pero preferimos utilizar el protocolo MQTT por las ventajas que se han comentado anteriormente, además de ser más complicado de integrar en la arquitectura de nuestro proyecto, por ser un servicio externo y dependiente de terceros.

Podemos encontrar en la red multitud de plataformas para el montaje de dashboards, tanto gratuitas como de pago, orientadas a la monitorización y al Internet de las Cosas. Así dimos con algunos como Plotly[1] , el cual es bastante potente en su versión de pago pero no tanto en la gratuita; Graphite[2], que aunque promete ser muy funcional aún está en fases tempranas de desarrollo; o Freeboard [3], sencillo, intuitivo y fácil de utilizar pero que en su

[1] https://plot.ly/
[2] https://github.com/graphite-project
[3] https://freeboard.io/

versión gratuita no permite tener dashboards privadas.

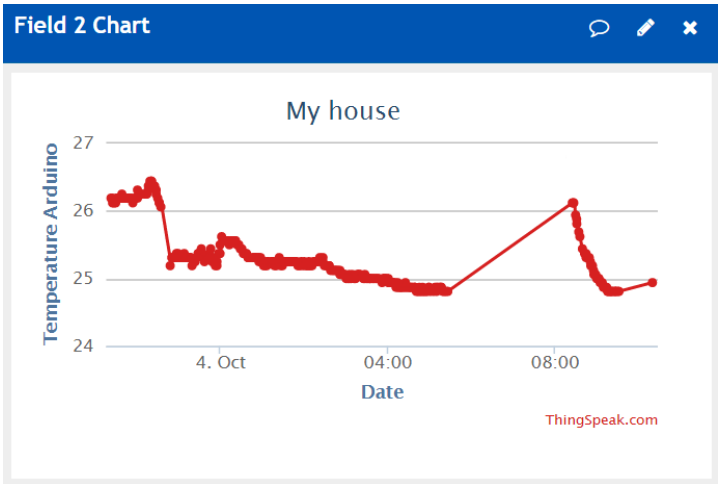


Figura 4.1: Ejemplo de funcionamiento de la aplicación ThingsPeak.

Después de leer foros de comunidades interesadas en IoT, viendo como la gente hablaba maravillas de cierta plataforma que, pese a estar por aquel entonces aún en desarrollo, tenía muchísima funcionalidad nos decidimos por Grafana, cuyas características se detallan a continuación.

4.1. Grafana

Grafana[4] es un visualizador en tiempo real de series temporales de datos que permite el uso de gráficos totalmente interactivos y editables, así como un sistema de alertas fácilmente configurables desde la interfaz gráfica de usuario. Otra de las utilidades que posee esta plataforma es un eficiente sistema de gestión de usuarios, mediante grupos y roles con distintos permisos.

[4] https://grafana.com/

Por último hay un banco bastante importante de de extensiones para Grafana cuya instalación es muy intuitiva y que permiten añadir funcionalidades como paneles con mapas para geolocalizar la fuente de nuestras métricas o APIs para integrar distintos tipos de bases o fuentes de datos.

4.1.1. Visualización de los datos

Grafana permite la visualizaión de los datos en formatos de gráfica muy variados con puntos, rayas, barras, etc. En nuestro caso y para series de datos temporales hemos considerado que lo mejor es mostrarlos de forma lineal como la gráfica de la Figura 4.2.

La configuración de las gráficas se hace mediante consultas InfluxDB como por ejemplo:

```
SELECT mean("value") FROM "mqtt_all" WHERE
"topic" = 'sensors/nodemcu/luminosidad'
```

De esta forma se puede mostrar en una gráfica todos los valores que entran con un determinado topic.

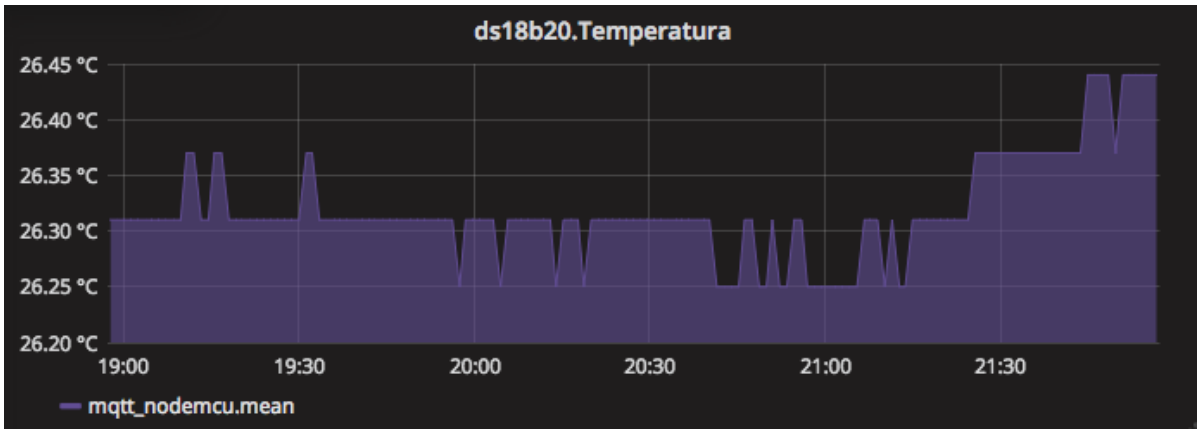


Figura 4.2: Visualización de la temperatura mediante líneas.

4.1.2. Alarmas y notificaciones

Una de las funcionalidades que nosotros consideramos más útiles en nuestro proyecto, es el soporte para alertas que ofrece Grafana. Éstas pueden configurarse desde la interfaz gráfica y permite el envío notificaciones no sólo por **email** si no por otras plataformas como **Slack** o **PagerDuty**.

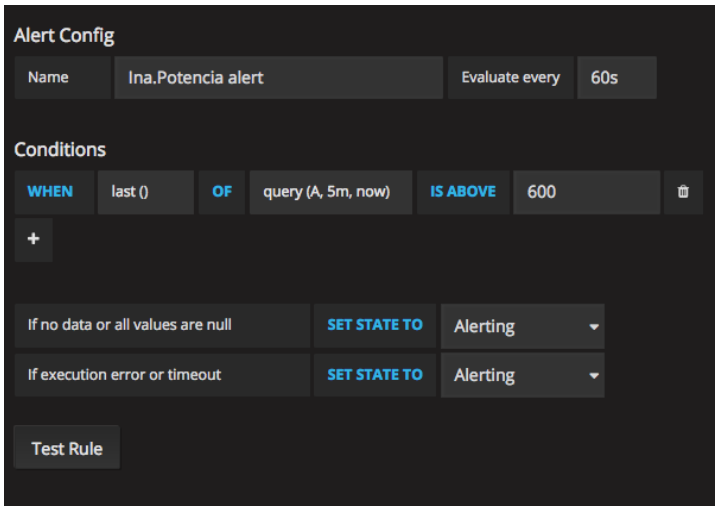


Figura 4.3: Pantalla de configuración de alertas en una gráfica.

El sistema permite definir las alertas, que funcionan como un trigger en la base de datos InfluxDB, de diferentes maneras (Figura 4.3). Así podemos encontrarnos con consultas de diferentes tipos para que la aplicación nos avise cuando un parámetro supere o baje de un determinado valor, cuando se deje de recibir dicho parámetro o consultas más complejas cómo por ejemplo que la media de los datos obtenidos en los últimos 5 minutos pase de un valor crítico. Podemos saber que gráficas tienen alertas configuradas, como la de la Figura Figura 4.4.

Para ponerla en funcionamiento tuvimos que crear una cuenta de correo con el nombre

grafana.alerts@gmail.com y especificamos la configuración SMTP (del servicio gMail) en el fichero de configuración de Grafana.

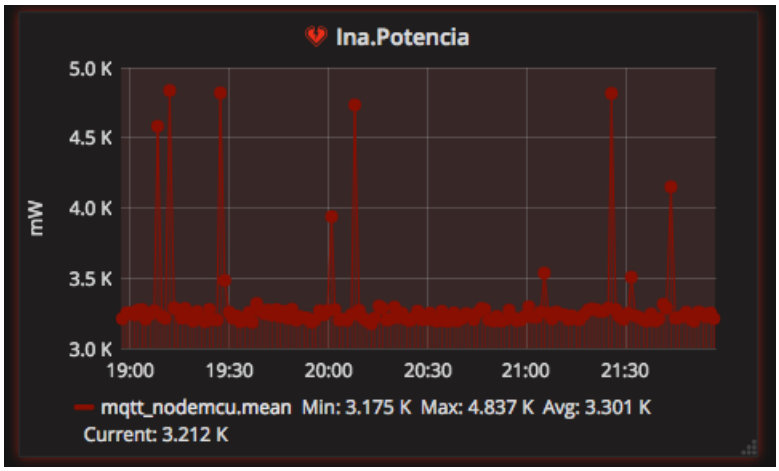


Figura 4.4: Cuando una gráfica tiene una alerta configurada se muestra un icono con un corazón junto al nombre de la misma.

4.1.3. Plugins, extensiones y APIs

Grafana ofrece una lista de extensiones[5], que va aumentando a medida que distintos desarrolladores o empresas publican sus trabajos en la comunidad de usuarios. Podemos encontrar de varios tipos: diferentes formatos de visualización de gráficas, APIs para conectar con distintas bases de datos, aplicaciones independientes de Grafana, etc.

4.2. Despliegue en la nube

A la hora de instalar los servicios se optaron por distintas alternativas, pero los tutores del proyecto nos facilitaron una máquina virtual, al que se puede acceder desde la dirección `pera.dacya.ucm.es`. Está máquina cuenta con el sistema operativo **Ubuntu 16.04.1** y las siguientes características hardware virtualizadas:

[5] `https://grafana.com/plugins`

- 1GB de memoria RAM.

- 14GB de capacidad de disco duro.

- Procesador Intel Westmere, que funciona a 3GHz, con una memoría caché de 4 MB y que sólo utiliza un core.

Por último y una vez teníamos el proyecto funcionando completamente, se decidió hacer una configuración similar a la que había instalada en el servidor, pero utilizando tecnologías de Cloud Computing, analizándose las siguientes plataformas para el despligue de la misma:

- ***Windows Azure.*** Es una plataforma de nube abierta y flexible que permite compilar, implementar y administrar aplicaciones rápidamente en una red global de centros de datos administrados por Microsoft. Puede compilar aplicaciones en cualquier lenguaje, herramienta o marco.

- ***Amazon Elastic Compute Cloud (EC2).*** Forma parte del conjunto de aplicaciones conocidas como Amazon Web Services. Proporciona capacidad informática con tamaño modificable en la nube. Amazon EC2 presenta un auténtico entorno informático virtual, que permite utilizar interfaces de servicio web (Figura 4.5) para iniciar instancias con distintos sistemas operativos, cargarlas con su entorno de aplicaciones personalizadas, gestionar sus permisos de acceso a la red y ejecutar su imagen utilizando los sistemas que desee.

Debido a que Amazon nos ofrecía más servicios gratuitos así como crédito para gastar en su plataforma por darnos de alta en la misma como estudiantes universitarios, decidimos decantarnos por ésta. Además de ser más segura puesto que genera un fichero de clave para que sólo se pueda conectar de forma segura mediante TLS.

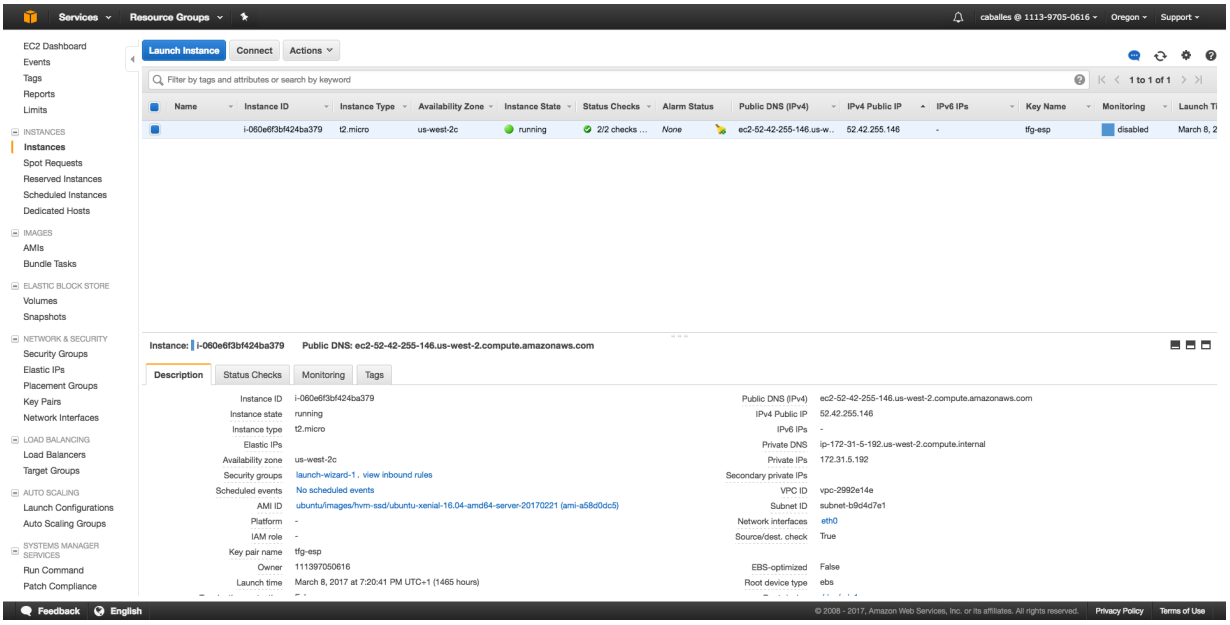


Figura 4.5: Panel de administración de Amazon E2C con la instancia de la máquina creada para el proyecto.

4.3. Casos Prácticos

Dada la flexibilidad de nuestro proyecto se han encontrado diferentes aplicaciones de uso del mismo, algunas de las cuales ya están en funcionamiento y otra está proyectada para un futuro a corto plazo.

4.3.1. Monitorización de sala fría

Siendo el objetivo inicial la monitorización de un CPD de grandes dimensiones no podíamos dar por finalizado el proyecto sin el despligue del mismo en una *sala fría* con un gran número de servidores, más concretamente uno de nuestros Nodos sensores se encuentra instalado en el laboratorio del Departamento de Arquitectura de Computadores y Automática en la Facultad de Físicas de la Universidad Complutense (Figura 4.6).

Se requería en este caso la medición de la temperatura, humedad y luminosidad (a mo-

do de sensor de presencia) de dicha sala, por lo que se han configurado sensores DHT22, DS18B20 (a modo de respaldo) y una fotoresistencia.

Figura 4.6: Nodo sensor instalado en sala fría.

Todas las gráficas se han configurado con alertas para asegurar el correcto funcionamiento del sistema de refrigeración de la sala, tal como puede verse en la Figura 4.7. La figura muestra el estado de la monitorización de la sala durante un período de siete días, reportando temperatura (paneles superior izquierdo e inferior), humedad relativa (panel superior derecho) y luminosidad (panel central). Tanto en los paneles de temperatura como de luminosidad se han establecido las alarmas correspondientes, configuradas para enviar correos electrónicos al administrador del laboratorio al superar 25ªC y una luminosidad del 60 %. En los paneles, se pueden observar, mediante líneas verticales de color rojo y verde, los momentos en los que se han enviado avisos al superar el umbral establecido y al recuperar un valor aceptable. El sensor de luminosidad envía alarmas a modo de aviso de presencia en la sala. El sistema ha funcionado de forma estable durante un elevado número de días y

se ha mostrado robusto ante falta de conectividad con el punto de acceso inalámbrico.

Figura 4.7: Dashboard del Nodo sensor instalado en sala fría.

4.3.2. Monitorización en laboratorio de Física de Materiales

El objetivo de la instalación es en este caso la monitorización de procesos de investigación en el ámbito de la Física de Materiales. Concretamente, se ha realizado una instalación de sensor de temperatura en un laboratorio de la Facultad de Ciencias Físicas donde, periódicamente, se realizan experimentos en un horno de vacío a altas temperaturas; dichos experimentos deben monitorizarse periódicamente por dos razones:

- *Seguridad*: debido a las altas temperaturas generadas en el proceso, los investigadores requieren un sistema para monitorizar la temperatura máxima alcanzada que implemente alertas al superar ésta un cierto umbral (fijado en este caso a 200^{o}C en la superficie exterior del horno). Hasta ahora no disponían de un sistema de monitorización de estas características.

- *Validación del experimento*: un experimento sólo será válido en el caso de haberse mantenido la temperatura por encima de cierto umbral durante todo el período experimental. El análisis a posteriori de los datos almacenados en la base de datos y visualizados a través del dashboard permite verificar estas condiciones.

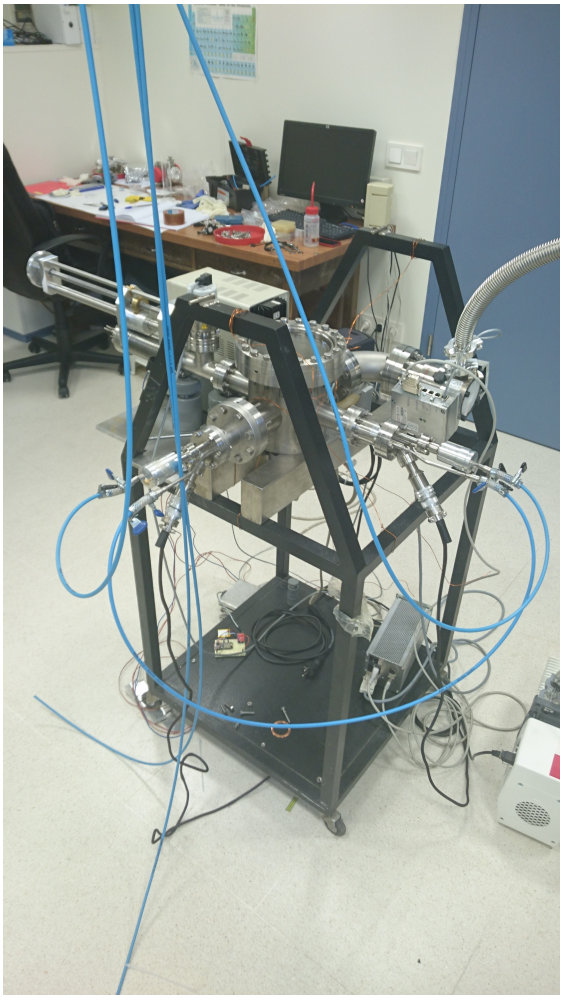


Figura 4.8: Instalación del termopar en el horno de laboratorio.

Debido al alto rango de temperaturas requerido se ha optado por un termopar MAX31855K con comunicación SPI y soporte para temperaturas mínima y máxima de entre 0^{o} y 400^{o}. Paralelamente, se ha desarrollado un dashboard de monitorización y alarmas exclusivo para

el personal de laboratorio. La disposición de ambos elementos puede observarse en las Figuras 4.8 y 4.9.

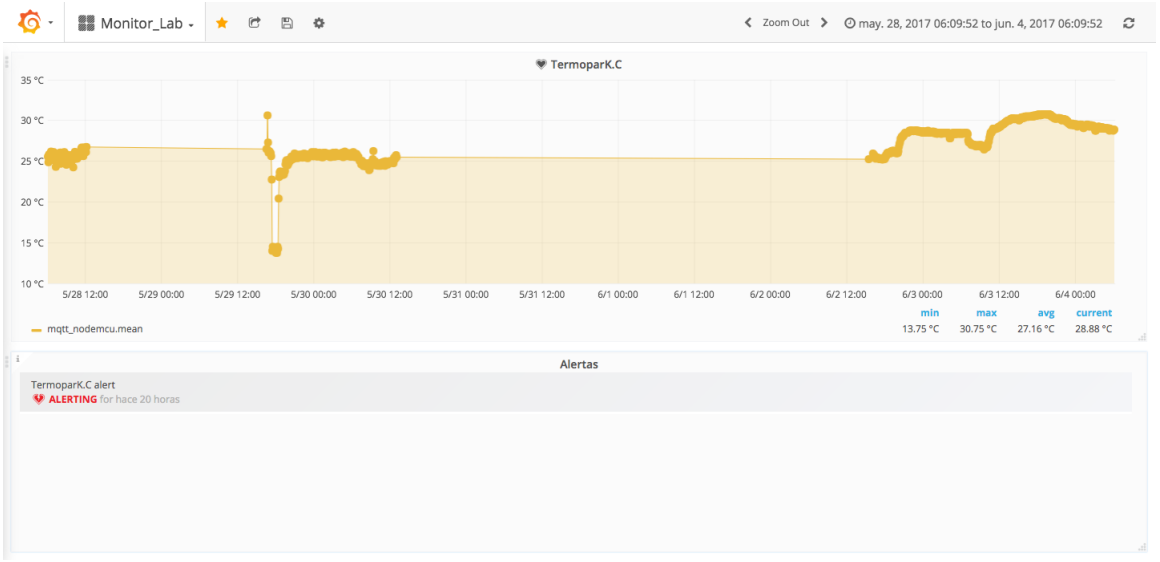


Figura 4.9: Dashboard del Nodo sensor instalado en horno de laboratorio.

Debido a la utilidad de la herramienta, los miembros del laboratorio han solicitado la ampliación del sistema para la monitorización de las condiciones de vacío durante el experimento, trabajo que se abordará en el futuro.

4.3.3. Monitorización del sistema de refrigeración en Facultad de Ciencias Físicas

Este caso real aún no se ha implementado todavía, pero se espera que pueda realizarse en un futuro próximo. El sistema de refrigeración de la facultad está formado por un circuito hidráulico que recorre la misma mediante tuberías.

Se pretenden instalar varios Nodos con algún sensor de temperatura (seguramente DS18B20) en el circuito, que envíen los datos a distintas gráficas del mismo dashboard para intercep-

tar picos o caídas de temperatura que nos puedan estar indicando fallos en el sistema de refrigeración de forma localizada.

Capítulo 5

Conclusiones

Se proponía al comienzo de este proyecto el desarrollo de un sistema capaz de monitorizar diferentes parámetros físicos relevantes en un **Centro de Procesamiento de Datos** de grandes dimensiones, lo cual se ha conseguido al disponer de las siguientes características:

- Se capturan los datos de temperatura, humedad, corriente, o luz, aunque fácilmente se podría reprogramar para obtener otras magnitudes puesto que se han utilizado varias interfaces de comunicación.

- Los datos son enviados por MQTT de forma segura y en el servidor se tratan y se almacenan en una base de datos que ofrece un gran rendimiento para sistemas de estas características.

- Los datos se muestran a través de una aplicación web, accesible desde cualquier dispositivo con conexión a internet. Además se ofrece un sistema de alertas vía correo electrónico para notificar de valores que no entran dentro de unos parámetros seguros.

Como se introdujo en el Capítulo 1 que el objetivo principal planteado consistía en conseguir un sistema barato, robusto y escalable, lo cual se ha conseguido puesto que:

- El precio del Nodo Sensor, contando con el controlador y sensores es inferior a 30€.

- Al haber probado baterías de respaldo, está protegido contra hipotéticos cortes de luz. Además después de haber testado los servicios de cloud computing de Amazon, se ha comprobado que se puede instalar todo el sistema con relativa facilidad en un servidor externo de alta fiabilidad y disponibilidad.

- Es fácilmente escalable ya que se puede ampliar el número de nodos sensores con el único límite de las direcciones IPs disponibles en la red, puesto que se puede reutilizar el mismo código para programar todos los controladores.

Además de todo esto, se puede afirmar que el sistema ofrece una gran versatilidad, ya que su funcionamiento es fácilmente extrapolable a otros ecosistemas, como la monitorización de un sistema de calefacción o de los parámetros físicos de un invernadero.

5.1. Conocimientos adquiridos y usados

Hemos utilizado diversos conocimientos relativos a varias ramas de la informática y de las asignaturas estudiadas durante el transcurso del Grado:

- Los lenguajes de programación LUA y Python eran desconocidos para nosotros; no obstante con la base que tenemos a lo largo de la carrera resultó relativamente sencillo adquirir unas nociones básicas con las que manejarnos con ellos. El código que usa Arduino IDE, prácticamente igual a C no nos supuso ningún problema al estar muy familiarizados con este lenguaje; además, en las asignatura **Programación de Sistemas y Dispositivos** y **Sistemas Empotrados** ya programamos periféricos sobre un microcontrolador, por lo que teníamos cierta experiencia en este ámbito.

- Por el lado de las comunicaciones entre Nodo y Broker, nos fueron de gran ayuda los conocimientos adquiridos en las asignaturas **Redes** y **Ampliación de Redes** a la hora de elegir protocolos a nivel de aplicación y transporte. También la asignatura optativa

Seguridad en Redes, así como temas tratados en **Ética, Legislación y Profesión** nos empujaron a encriptar los mensajes mediante TLS para que el intercambio de información fuera lo más seguro posible.

- Tampoco teníamos ninguna experiencia previa en bases de datos no relacionales, por lo que fue un tema totalmente novedoso. Afortunadamente, el formato de consultas utilizado por InfluxDB es similar a SQL, sí estudiado en la asignatura **Bases de Datos** y por tanto la adaptación no resultó complicada.

- No nos resultó complicado utilizar el servidor Linux puesto que estamos muy acostumbrados a trabajar en este Sistema Operativo, que estudiamos con bastante profundidad en **Sistemas Operativos** y **Ampliación de Sistemas Operativos**.

- Por último, la asignatura **Ingeniería del Software** nos dio unas nociones sobre cómo se debe planificar y gestionar un proyecto.

Por otra parte los trabajos presentados a lo largo de toda la carrera, así como las diferentes presentaciones que hemos realizado en todas las asignaturas nos han venido muy bien a la hora de redactar la memoria y planificar la presentación de la misma.

5.2. Desafíos encontrados

El primero de los problemas encontrado fue el desarrollo inicial con Lua. Si ya de por sí el lenguaje nos resultaba nuevo pronto nos topamos con que, al intentar que el controlador entrara en estado DeepSleep entre lecturas y envíos, éste no volvía a funcionar correctamente. Investigando por foros descubrimos que se trataba de un problema de las librerías del lenguaje por lo que, entre otros motivos, renunciamos a seguir avanzando en dicho framework.

No fue el único problema encontrado en relación a DeepSleep, puesto que una vez terminado el proyecto y ya en fase de investigación de posibles mejoras, intentamos añadir funcionalidades con comunicación bidireccional, haciendo que el sensor pudiera recibir mensajes del servidor. No obstante, al estar la placa dormida no se recibían esos mensajes y se perdían. Por lo que en un futuro y si se pensase en añadir esta funcionalidad habría que pensar si interesa aumentar el consumo, sobre todo si el sistema trabaja de forma autónoma conectado a una batería.

Por último, otro de los desafíos que nos encontramos fue el hacer funcionar el sistema de alertas o alarmas. Cuando nos decidimos a usar Grafana en los foros de desarrolladores se comentaba que se estaba trabajando en este sistema, y prácticamente el día que se publicó la primera beta nosotros comenzamos a testearla, sin apenas documentación ni referencias.

El usuario que habíamos asignado a Grafana en InfluxDB sólo tenía permisos de lectura, pero al funcionar las alertas como un trigger en la base de datos no podía ejecutarse al no tener permisos, detalle que conllevó cierto retraso en el desarrollo. La configuración del protocolo SMTP para el envío de correos electrónicos de alerta conllevó también cierto esfuerzo adicional.

5.3. Posibles mejoras y objetivos futuros

Una de las primeras mejoras a implementar, cuya viabilidad ya se ha probado, es el despliegue de toda la infraestructura del servidor utilizando **Cloud Computing**. Plataformas como Azure de Windows y EC2 de Amazon Web Services podrían resultar económicas y totalmente factibles.

Otro de los objetivos marcados a corto plazo es la instalación y configuración de diferentes

sensores como pueden ser proximidad, humo, presencia para poder utilizar el proyecto en diferentes ámbitos y ecosistemas de trabajo. Se nos ha propuesto el montaje del sistema en un invernadero, en la monitorización de un sistema de calefacción o el control de la temperatura de un horno.

5.4. Aportación individual de los miembros del grupo al proyecto

Por lo general el trabajo se ha desarrollado de forma conjunta, organizando sesiones conjuntas para implementar de manera cooperativa los distintos aspectos del proyecto.

Al comienzo del proyecto, mientras se evaluaban los distintos lenguajes de programación que se podían usar para configurar los nodos, Denys se encargó de probar más en profundidad **MicroPython** mientras Carlos investigaba con **Lua** para, después de poner nuestras respectivas conclusiones en común, programar de forma conjunta un primer script en Lua.

Una vez decididos por **Arduino** por los motivos explicados en los apartados anteriores cada uno de nosotros comenzó el trabajo sobre una placa diferente (NodeMCU Denys y Feather Huzzah Carlos) para, de forma paralela, ir programando el funcionamiento de diferentes sensores mientras cada uno investigábamos diferentes tecnologías que se podrían usar en el servidor. Reuniendo cada dos semanas para poner en común nuestros avances y la información descubierta.

Además en estas reuniones periódicas se aprovechó para instalar y configurar todos los parámetros necesarios para hacer funcionar los programas en el servidor.

Bibliografía

[1] INA219 - DataSheet.

https://cdn-shop.adafruit.com/datasheets/ina219.pdf.

[2] DHT22 - DataSheet.

https://www.sparkfun.com/datasheets/Sensors/Temperature/DHT22.pdf.

[3] DS18B20 - DataSheet.

https://cdn.sparkfun.com/datasheets/Sensors/Temp/DS18B20.pdf.

[4] MAX31855 - DataSheet.

https://datasheets.maximintegrated.com/en/ds/MAX31855.pdf.

[5] NodeMCU - Documentación.

https://nodemcu.readthedocs.io/en/dev/.

[6] Arduino - Documentación.

https://www.arduino.cc/.

[7] FeatherHUZZAH - Documentación.

https://learn.adafruit.com/adafruit-feather-huzzah-esp8266.

[8] FeatherHUZZAH - Pinouts.

https://learn.adafruit.com/adafruit-feather-huzzah-esp8266/pinouts.

[9] NodeMCU - Pinouts.

https://github.com/opendata-stuttgart/meta/wiki/Pinouts-NodeMCU-v2,-v3.

[10] MQTT - Documentación.

http://mqtt.org/documentation.

[11] Mosquitto - Broker.

http://mosquitto.org/.

[12] Mosquitto - TLS.

https://mosquitto.org/man/mosquitto-tls-7.html.

[13] InfluxDB - BBDD.

https://docs.influxdata.com/influxdb/v1.2/.

[14] Telegraf - Collect.

https://docs.influxdata.com/telegraf/v1.2/.

[15] Grafana - Dashboard.

https://grafana.com/.

[16] Thingspeak - Dashboard.

https://thingspeak.com/.

Apéndice A

Introduction

IoT (***Internet of Things***) is a paradigm that consists of interconnecting the objects of everyday life through the Internet. Through this paradigm, it is possible to deploy large networks of sensors that monitor, in real time and jointly a multitude of parameters and can, if necessary, act on the observed values and based on predefined parameters or thresholds.

The IoT concept was first presented by Kevin Ashton in 1999 at the MIT, but its popularity has only risen in recent years mainly due to the boom in the smart phones technology. According to Gartner, in 2020 there will be 12.863 million of things connected[1], mainly the Home Automation Market devices. It will allow the users to control a series of integrated devices such as air conditioning systems, security infrastructures and multimedia tools.

A.1. Aims and system overview

The main purpose of this project is to create a low price, robust and easily escalated system to monitor in real time physical factors that might be relevant in a Data Processing Center, such as humidity, temperature, consumption or light.

[1] http://www.gartner.com/newsroom/id/2636073

In addition, it is desirable that the developed system may be extended to other areas in which the monitoring of any relevant process is crucial, without impact on the end user. Given the versatility of the visualization tool *dashboard* used, it allows to select in a very intuitive way what data we want to see and how to do it, as well as to program alarms that notify us when a parameter exceeds the values that we indicate and the multitude of sensors that can be connected to the node. It is possible to find other utilities and practical cases as we will see later, see when monitoring the refrigeration circuit of the Faculty of Physical Sciences or monitor the correct operation of vacuum furnaces of the Physics laboratory of Materials.

Figure A.1 illustrates a scheme that serves to make us the idea of the general structure of the proposed system.

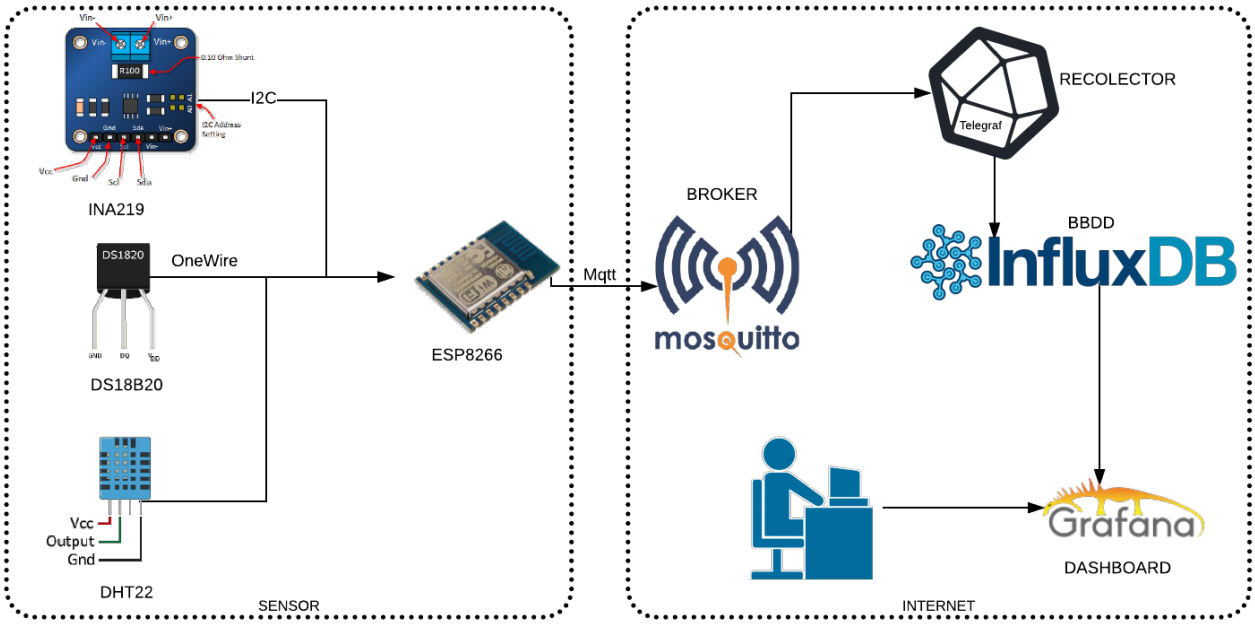


Figura A.1: General scheme of the proposed infrastructure.

A.2. Initial requirements

For the development of the system, it is intended to create a project that meets, at least, the following needs:

A.2.1. Sensor node

1. It is intended that the device has a **low cost**, in order to be easily installable with a minimum investment.

2. We also want **low consumption**, since it may be necessary to install a large number of devices in the same power grid and even that these are powered by batteries.

3. It is obvious that, since it can measure critical parameters, it has to be a robust system, so we need **good software support (firmware)**.

4. Being a fairly versatile project, which can be mounted in many scenarios, we need the sensor node to be **compatible with different sensor types and interfaces**.

5. Finally, it is recommended that the node we choose has a **good support and community level** to solve problems quickly and accurately.

A.2.2. Data persistence

Another needs, which we find is that you can consult immediately a record of the data collected by the sensor nodes, so be valued different systems databases so that this information storage is done in the most efficient way possible.

A.2.3. Two-way communication

It may be the case that we need to communicate with the sensor node remotely. To do this, it is necessary to establish a mechanism for receiving messages in the same so that, if necessary, we can give orders or transmit information to the microcontroller.

A.2.4. Visualization and alerts

Obviously this whole system would not make sense if you could not consult the data obtained in an easy and intuitive way. So you need to look for any dashboard or display platform that meets those requirements. It is also a recommended requirement that, somehow, we will be notified when the parameters collected by the sensors exceed critical values.

A.2.5. Security

Finally, if we get the communication between the microcontroller and the Broker to be done safely (with some encryption protocol and/or authentication) and protect the access to the different services, we can avoid security problems and information theft.

A.3. Technologies and resources used

Given the above requirements, the IoT system is developed in the following four main layers: the device, communication interface, persistence and visualization, whose functionality and characteristics are introduced below and will be detailed in the rest of the document:

- ***The device.*** It is responsible for data gathering, in this case, one low cost Node ESP8266 implemented in different boards (NodeMCU[5] or Feather HUZZAH[7]) to which is connect different sensors such as INA219[1], DS18B20[3] or DHT22[2].

- ***Communication interface/gateway.*** This part is in charge of receiving and treating the information sent by the Nodes. It is implemented in an Ubuntu server that has configured services such as Mosquito[11] (receives messages by MQTT[10]) and Telegraf[14] (treat the data in those messages).

- ***Persistence.*** It is the system main feature, it allows its administration in an intelligent

way. It is managed through InfluxDB[13], which is responsible for the storage of the data sent by the Nodes.

- ***Visualization.*** The data is finally represented in a dashboard using Grafana [15] technology, which allows multiple customizations options in order to provide the final information in an easy and friendly tools that also has alarm options.

A.4. Methodology and work plan

The purpose of this project is to build a real-time monitoring infrastructure based on IoT, selecting, integrating and configuring already developed tools, and implementing the functionality required in the sensor node through the corresponding firmware.For this the work has been divided in two parts, one of investigation and test and one of development.

It starts by evaluating different **frameworks** and programming languages for Sensor Nodes, testing up to three languages with their respective IDEs. Once decided by the **Arduino**[6] environment for its support through Internet communities, the multitude of libraries developed and familiarity with the programming language C, as well as for the stability observed when using it on the board NodeMCU. Once decided, we began to value technologies to configure the services of the server. This point is mainly a research work through specialized forums in IoT, blogs or comparative pages.

Once all this is decided in parallel, we work on the programming of the Sensor Nodes and the installation and configuration of the different services of the server.

With a completely stable and functional system the work has consisted of a final phase of evaluation of different ways to improve infrastructureuse of Cloud Computing in domains

such as *AWS* or *Azzure*, bidirectional communication between server and sensor node or add new sensors to measure new or existing parameters.

A.5. Structure of the document

The present report it has been divided into chapters according to the following structure:

- Chapter 1 discusses an introduction to the project, explaining the objectives and starting requirements, the technologies and the work plan that will be used to reach them.
- Chapter 2 talks about everything about the Sensor Node. From the physical and electronic part of it, to the software platforms that have been evaluated and used for the programming of the same, passing through the different sensors used and the communication interfaces they use.
- Chapter 3 explains the sending and receiving of messages between the microcontroller and the server in a secure way, including their treatment and storage in the database..
- Chapter 4 deals with the deployment of different services in the cloud, and the visualization of the data in the dashboard, as well as its alarm system. In addition, real practical cases where the project has been implemented are discussed in this section.
- Chapter 5 discusses the conclusions reached after developing our system, comparing them with the initial requirements and mentioning the knowledge and skills provided by the system.

Apéndice B

Conclusions

It was proposed at the beginning of this project the development of a system capable of monitoring several relevant physical parameters in a large **Data Processing Center**, which was achieved by obtaining the following features:

- It captures temperature, humidity, current, or light data although it could easily be reprogrammed to obtain other magnitudes since the development required several communication interfaces.
- The information is sent by MQTT securely and on the server this data is processed and stored in a database that offers a great performance for systems with these characteristics.
- The data is shown through a web application, accessible from any device with internet connection. Besides it offers a system of alerts via email to notify of values out of safe parameters.

As commented on Appendix A, the main aim was to achieve a low price system, robust and scalable, which has been achieved:

- The sensor node price plus the controller and sensors is less than 30€.

- The backup batteries are protected against hypothetical courts of light. Besides after testing the Amazon cloud computing services, it was proven that it can all be installed with relative ease in an external server of high reliability.

- It is easily scalable since you can expand the number of sensor nodes with the only limit of the IPs addresses directions available in the network, since you can reuse the same code to program all the controllers.

In addition, the system offers a huge versatility since its operation can be easily transferred to other ecosystems, like the monitoring of a heating system or the physical parameters of a greenhouse.

B.1. Acquired and used knowledge

We have used knowledge of several computing branches as well as knowledge from the subjects studied in the career.

- We did not know at first the programming languages LUA and Python, nevertheless with the base that we have obtained along the career, it did not cost us much to learn some basic notions for both languages The Arduino IDE code is practically the same used in C, so it was no problem for us since we are very familiarized with this language, besides that in the subject "Programación de Sistemas y Dispositivos" and "Sistemas Empotrados" we had already programed peripheral on a microcontroller, so we had some experience in this field.

- Regarding the communications between Node and Broker, it was a great help for us the knowledge obtained in the subjects "Redes y Ampliación de Redes" in order to choose protocols for application and transport. Also the "Seguridad en Redes", as well as subjects treated in "Ética, Legislación y Profesión" pushed us to encrypt the messages by means of TLS so that the exchange of information was the safest possible.

- Neither had we had any previous experience in no relational databases, it was a complete new ground for us. Luckily, the format of queries that it is used in InfluxDB is quite similar to SQL, which we have already studied in the subject "Bases de Datos", so it was not difficult to become fluent in this language.

- It was not complicated to use the Linux server since we are very used to work with Operative Systems, which we study in the subject "Sistemas Operativos y Ampliación de Sistemas Operativos".

- Finally, the subject "Ingeniería del Software" gave us some notions on how to schedule and manage a project.

On the other hand, the deliverables handled along all the career, as well as the different presentations that we have made in all the classes have helped us elaborate the documentation and plan its presentation.

B.2. Challenges found

The first problem that we faced was when we started to develop with Lua. The language itself was very new to us and we soon faced a problem when we tried to put the controller on DeepSleep mode between readings and it did not work properly afterwards. After investigating in some forums, we discovered that is was a problem with the languages libraries so among other reasons, we refused to keep advancing with this framework.

It was not the only problem that had with the DeepSleep, since once we had finished the project and decide to investigate improvements, we tried to add functionalities with bidirectional communication so the sensor could receive messages from the server. However, because the plate was asleep it did not receive these messages and they got lost. Having

said that, in the future if this functionality was to be implemented it would be first recommended to analyze weather the increase in consumption, especially if the system works of autonomous form connected to a battery.

Finally, another of the challenges that we faced was to make work the alerts and alarms system. When we decide us to use Grafana, at the developer's forums they were discussing working with this tool and basically the day that the first beta version was released we started testing it, without any documents nor references.

The user that had assigned to Grafana in InfluxDB only had reading permissions, but since the alerts worked like a trigger in the database, it could not be executed due to not having permissions. We had lots of problems with this matter up until we found out what was causing the error. In addition to it, with the first settings files there were no clear specification regarding the configuration SMTP for the mailing.

B.3. Improvements and future aims

One of the first improvements to implement, whose feasibility has already being tested, is the deployment of all the server infrastructure using Cloud computing. Platforms like Azure of Windows and EC2 from Amazon Web Services could result economic and totally feasible.

Another of the aims marked in the short term is the installation and configuration of different sensors as they can be vicinity, smoke, presence to be able to use the project in different fields and ecosystems of work. It has proposed to us the setting of the system in a greenhouse, for monitoring a heating system or the oven temperature control.

B.4. Individual contribution of the members of the group to the project

Generally the project has been developed by the group jointly, we gathered to implement in a collaborative way the different tis project's steps.

At the beginning of the project, while we evaluated the distinct programming languages that could be used to configure the nodes, Denys was responsible to test in depth **MicroPython** while Carlos investigated with **Lua** in order to later, after sharing our perspectives and conclusions, we stated to program together the first script in Lua.

Once we decided to use **Arduino** due to reasons explained in the previous sections, each one of us carried a plate home (NodeMCU Denys and Feather Huzzah Carlos) in order to work in parallel to program the operation of different sensors while each one investigated different technologies that could be used in the server. We gathered each two weeks to put in common our advances and the information discovered.

At these periodic meetings we install and configure all the necessary parameters to do work the programs in the server.

Apéndice C

Instrucciones de instalación

Se proporcionan, como documentación adicional, las instrucciones y pasos básicos necesarios para la instalación y configuración de los distintos servicios utilizados para el desarrollo del proyecto.

C.1. InfluxDB[1]

En desde la terminal y en el directorio *home* ejecutamos:

```
$ wget https://dl.influxdata.com/influxdb/releases/influxdb_1.2.1_amd64.deb
$ sudo dpkg -i influxdb_1.2.1_amd64.deb
```

Después de la instalación podremos editar la configuración de la base de datos InfluxDB ubicada en el fichero ***/etc/influxdb/influxdb.conf***. A continuación podremos acceder a la misma desde cualquier navegador web, para ello ello debemos abrir la URL de InfluxDB, que será la dirección ip del servidor en el que se ha instalado, accediendo desde el puerto 8083: ***http://ip:8083***.

[1]https://docs.influxdata.com/influxdb/v1.2/introduction/installation/

C.2. Telegraf[2]

Al igual que en el paso anterior, se ejecutan desde la consola de comandos ubicados en el directorio *home* las siguientes instrucciones:

```
$ sudo wget https://dl.influxdata.com/telegraf/releases/telegraf_1.2.1_amd64.deb
$ sudo dpkg -i telegraf_1.2.1_amd64.deb
```

Una vez completada la instalación, desde el fichero de configuración ***/etc/telegraf/telegraf.conf*** se pueden insertar todos los parametros que nos hacen falta, como fuentes de datos de entrada y de salida. Por ejemplo la URL de nuestro servidor InfluxDB se indica en la sección llamada ***outputs.influxdb***.

C.3. Grafana[3]

En *home* ejecutamos desde la línea de comandos:

```
$ sudo wget https://grafanarel.s3.amazonaws.com/builds/
grafana_4.1.2-1486989747_amd64.deb
$ sudo dpkg -i grafana_4.1.2-1486989747_amd64.deb
```

Una vez se ha iniciado el servicio sin problemas, ya podemos ir a nuestra URL por el puerto 3000 ***http://ip:3000*** desde un explorador web y entrar a la aplicación con el usuario por defecto (*user: admin, password: admin*) para configurar desde la interfaz gráfica el resto de parámetros como dashboards, alarmas, usuarios y permisos, etc.

[2]https://docs.influxdata.com/telegraf/v1.2/introduction/installation/
[3]https://docs.grafana.org/installation/

C.4. Mosquitto[4]

Mosquitto es el ***Broker*** que vamos a utilizar para la recepción y manejo de los mensajes MQTT de nuestro sistema. Para instalarlo sólo tenemos que ejecutar en el directorio *home* la siguiente instrucción:

```
$  sudo apt-get install mosquitto
```

Para cambiar los diferentes parámetros del servicio como pueden ser autentificación, puertos, certificados de encriptación, etc., iremos al fichero de configuración ubicado en ***/etc/mosquitto/mosquitto.conf***.

Además, la herramienta *Mqtt-Spy* es una aplicación que puede ser útil para monitorizar la actividad del sistema en caso de dudar del correcto funcionamiento del mismo.

C.5. Ejecución

Para ejecutar hemos de cargar el programa del Apéndice D en la ***ESP8266*** usando el Arduino IDE [5]. Después hemos de levantar InfluxDB, Mosquitto y Grafana. Una vez que todos los sistemas están arrancados y funcionando accedemos a Grafana por la siguiente url ***http://ip:3000*** y configuramos el origen de los datos como InfluxDB.

[4]https://mosquitto.org/documentation/
[5]https://www.arduino.cc/en/main/software

Apéndice D

Firmware Arduino

A continuación se adjunta el código en Arduino que se ha utilizado para la instalación de todos los sensores y la inicialización de los distintos buses e interfaces que se han mencionado y explicado a lo largo del proyecto:

```
//** Librerías **//
#include <OneWire.h>              // OneWire
#include <DallasTemperature.h>    // DS18B20
#include <ESP8266WiFi.h>          // WiFi
#include <ESP8266WebServer.h>     // TCP
#include <MCP3008.h>              // ADC
#include <PubSubClient.h>         // MQT
#include <DHT.h>                  // DHT
#include <Wire.h>                 // INA219
#include <WiFiClientSecure.h>     // Cliente WiFi con soporte de TLS
#include <Adafruit_INA219.h>      // INA219
#include "FS.h"                   // File System

//** Definiciones  **//
/* GPIOS: En esta región se definen distintos GPIOS que se
   usaran para establecer la comunicación con los periféricos. */
//SPI
#define CS_PIN 14
#define CLOCK_PIN 5
```

```
20 #define MOSI_PIN 13
21 #define MISO_PIN 12
22 //Fin SPI
23 #define DS18B20_PIN D5
24 #define DHT_PIN D3
25 #define DHT_TYPE DHT22
26 #define LDR_PIN A0
27 #define LED_PIN D4
28 /* Fin GPIOS */
29
30 /* Variables & Funciones */
31 // Función donde se trataran los mensajes MQTT recibidos por Nodo
32 void mqtt_sus (char* topic, byte* payload, unsigned int length);
33
34 const char* ssid =  "ssid";
35 const char* pass =  "pass";
36 const int   mqtt_portSSL = 8883; //mqtt_port = 1883;
37 const char* mqtt_server = "ip";
38 const char* mqtt_user = "user";
39 const char* mqtt_pass = "pass";
40
41 const int x = 1;
42 const int timeSleep = x * 60; // x = minutes
43 const float coeficiente_porcentaje = 100.0/1023.0;
44 // datos Ds18b20
45 char temperatureString[6];
46 // datos DHT22
47 char humidityDhtString[6];
48 char temeperatureDhtString[6];
49 // datos Ina219
50 char busVoltageString[6];
51 char power_mWString[6];
52 char current_mAString[6];
53 // datos LDR A0
```

```
54 char brightnessString[6];
55 // datos LDR MCP
56 char brightnessMcpString[6];
57 /* Fin Variables & Funciones */
58 /* ThingSpeak
59 const char* host = "api.thingspeak.com";
60 String ApiKey = "ZXKQ6RBIYQA6DZWG";
61 String path = "/update?key=" + ApiKey + "&field2=";
62 String pathhVol = "/update?key=" + ApiKey + "&field3=";
63 const int httpPort = 80;
64 */
65
66 /* Constructores */
67 Adafruit_INA219 ina219; // Bus: I2C
68 WiFiClientSecure wifiClient;
69 MCP3008 adc(CLOCK_PIN, MOSI_PIN, MISO_PIN, CS_PIN);
70 DHT dht(DHT_PIN, DHT_TYPE);
71 OneWire oneWire(DS18B20_PIN);
72 DallasTemperature DS18B20(&oneWire);
73 PubSubClient client(mqtt_server, mqtt_portSSL, wifiClient);
74 /* Fin Constructores */
75
76 //** Programa **//
77 void setup() {
78   // Configuramos el puerto serie
79     Serial.begin(115200);
80     Serial.println("");
81   // Configuramos la conexión WiFi
82     setup_wifi();
83   // Subimos el certificado para establecer la conexión segura
84     if (!SPIFFS.begin()) {
85       Serial.println("Failed to mount file system");
86       return;
87     }
```

```
88      File ca = SPIFFS.open("/ca.crt", "r");
89      Serial.println(ca.size());
90      if (!ca)
91        Serial.println("Error to open ca file");
92      else
93        Serial.println("Success to open ca file");
94
95      if(wifiClient.loadCertificate(ca))
96        Serial.println("loaded");
97      else
98        Serial.println("not loaded");
99    // Preparamos los sensores
100     dht.begin();
101     DS18B20.begin();
102
103     /*
104     also default ...
105     GPIO5 D1 SCL
106     GPIO4 D2 SDA
107     */
108     Wire.begin(D2, D1); // I2C -> sda, scl
109     ina219.begin();
110
111   pinMode(LED_PIN, OUTPUT);
112 }
113
114 // Función que realiza la conexión WiFi
115 void setup_wifi() {
116   delay(20);
117   // start wifi
118   WiFi.begin(ssid, pass);
119   while (WiFi.status() != WL_CONNECTED) {
120     delay(500);
121     Serial.print(".");
```

```
122   }
123   Serial.println("");
124   Serial.print("Connected to ");
125   Serial.println(ssid);
126   Serial.print("IP address: ");
127   Serial.println(WiFi.localIP());
128 }
129 
130 // Función que lee los datos del sensor DS18B20
131 float getTemperature() {
132   float temp;
133   do {
134     DS18B20.requestTemperatures();
135     temp = DS18B20.getTempCByIndex(0);
136     delay(200);
137   } while (temp == 85.0 || temp == (-127.0));
138   return temp;
139 }
140 
141 // Función que establece la conexión con el servidor MQTT
142 void reconnect() {
143   while (!client.connected()) {
144     Serial.print("Attempting MQTT connection...");
145     String clientId = "ESP8266Client-";
146     clientId += String(random(0xffff), HEX);
147     if (client.connect(clientId.c_str(), mqtt_user, mqtt_pass)) {
148       Serial.println("connected");
149       //client.subscribe("ledStatus");
150     } else {
151       Serial.print("failed, rc=");
152       Serial.print(client.state());
153       Serial.println(" try again in 5 seconds");
154       // wait 2 seconds before retrying
155       delay(2000);
```

```
156       }
157     }
158   }
159
160   // Bucle principal
161   void loop() {
162     Serial.println("");
163     // Conectamos al Broker MQTT
164       if (!client.connected()) {
165         reconnect();
166       }
167     // Led ON
168      digitalWrite(LED_PIN, HIGH);
169
170     // Leemos y preparamos el dato del adc_interno(conversor analógico digital)
171       float valueLdr = analogRead(LDR_PIN);
172       float brightness = valueLdr * coeficiente_porcentaje;
173       dtostrf(brightness, 2, 2, brightnessString);
174       // Lo enviamos por el protocolo MQTT
175       client.publish("sensors/nodemcu/luminosidad", brightnessString);
176       delay(100);
177
178     // Leemos y preparamos el dato del adc_MCP
179       valueLdr = adc.readADC(0);  // channel 0
180       brightness = valueLdr * coeficiente_porcentaje;
181       dtostrf(brightness, 2, 2, brightnessMcpString);
182       // Lo enviamos por el protocolo MQTT
183       client.publish("sensors/nodemcu/luminosidad",     brightnessMcpString);
184       delay(100);
185
186     // Leemos y preparamos los datos de ina219
187       float shuntvoltage = ina219.getShuntVoltage_mV();
188       float busvoltage = ina219.getBusVoltage_V();
189       float current_mA = ina219.getCurrent_mA();
```

```
190     float loadvoltage = busvoltage + (shuntvoltage / 1000);
191     float power_mW = (current_mA) * loadvoltage;
192     dtostrf(loadvoltage, 2, 2, busVoltageString);
193     dtostrf(power_mW, 2, 2, power_mWString);
194     dtostrf(current_mA, 2, 2, current_mAString);
195     // Los enviamos por el protocolo MQTT
196     client.publish("sensors/nodemcu/loadVolIna", busVoltageString);
197     delay(100);
198     client.publish("sensors/nodemcu/powerIna", power_mWString);
199     delay(100);
200     client.publish("sensors/nodemcu/currentIna", current_mAString);
201     delay(100);
202   // Leemos y preparamos el dato de ds18b20
203     float temperature = getTemperature();
204     dtostrf(temperature, 2, 2, temperatureString);
205     // Lo enviamos por el protocolo MQTT
206     client.publish("sensors/nodemcu/temperature", temperatureString);
207     delay(100);
208
209   // Leemos y preparamos los datos de dht22
210     float h = dht.readHumidity();
211     float t = dht.readTemperature();
212     if (isnan(h) || isnan(t)) {
213       Serial.println("Failed to read from DHT sensor!");
214     }
215     dtostrf(h, 2, 2, humidityString);
216     dtostrf(t, 2, 2, temeperatureDhtString);
217     // Los enviamos por el protocolo MQTT
218     client.publish("sensors/nodemcu/humidityDht", humidityDhtString);
219     delay(100);
220     client.publish("sensors/nodemcu/temperatureDht", temeperatureDhtString);
221     delay(100);
222
223   Serial.println("Closing MQTT connection...");
```

```
224     client.disconnect();
225
226     Serial.println("Closing WiFi connection...");
227     WiFi.disconnect();
228     delay(100);
229
230     // Configuramos el modo deepSleep
231       // WAKE_RF_DEFAULT, WAKE_RFCAL, WAKE_NO_RFCAL, WAKE_RF_DISABLED.
232       ESP.deepSleep(1000000 * timeSleep,WAKE_NO_RFCAL);
233
234 }
```

Apéndice E

Scripts LUA

Este código son los scripts LUA que fueron desarrollados como primera versión, pero desechada por problemas a la hora de configurar el modo DeepSleep. En ellos, vemos la conexión por WIFI, lecturas de los datos del sensor ds18b20 y el envío de mensajes a través de MQTT.

E.1. init.lua

```
1 app = require("application")
2 config = require("config")
3 setup = require("setup")
4 -- https://github.com/nodemcu/nodemcu-firmware/tree/master/lua_modules
5 -- DS18B20 one wire module for NODEMCU
6 ds18b20 = require("ds18b20")
7
8 setup.start()
```

E.2. setup.lua

```
local module = {}

local function wifi_wait_ip()
  if wifi.sta.getip()== nil then
    print("IP unavailable, Waiting...")
  else
    tmr.stop(1)
    print("\n====================================")
    print("ESP8266 mode is: " .. wifi.getmode())
    print("MAC address is: " .. wifi.ap.getmac())
    print("IP is "..wifi.sta.getip())
    print("====================================")
    app.satart()
  end
end

local function wifi_start(list_aps)
    if list_aps then
        --
        for key,value in pairs(list_aps) do
            if config.SSID and config.SSID[key] then
                wifi.setmode(wifi.STATION);
                wifi.sta.config(key,config.SSID[key])
                wifi.sta.connect()
                print("Connecting to " .. key .. " ...")
                --config.SSID = nil  -- can save memory
                tmr.alarm(1, 2500, 1, wifi_wait_ip)
            end
        end
    else
        print("Error getting AP list")
    end
```

```
33 end
34
35 function module.start()
36   print("Configuring Wifi ...")
37   wifi.setmode(wifi.STATION);
38   -- scan acces point
39   wifi.sta.getap(wifi_start)
40 end
41
42 return module
```

E.3. application.lua

```
1  local module = {}
2  m = nil
3
4  -- sends a simple ping to the broker
5  local function send_ping()
6      m:publish(config.ENDPOINT .. "ping","id=" .. config.ID,0,0)
7  end
8
9  -- sends my id to the broker for registration
10 local function register_myself()
11     m:subscribe(config.ENDPOINT .. config.ID,0,function(conn)
12         print("Successfully subscribed to data endpoint")
13     end)
14 end
15
16 local function mqtt_start()
17     m = mqtt.Client(config.ID, 120)
18     -- register message callback beforehand
19     m:on("message", function(conn, topic, data)
20       if data ~= nil then
21         print(topic .. ": " .. data)
```

```
22          -- do something, we have received a message
23          -- execute_command(data)
24        end
25      end)
26      -- connect to broker
27      m:connect(config.HOST, config.PORT, 0, 1, function(con)
28          register_myself()
29          -- cnd then pings each 1000 milliseconds
30          tmr.stop(6)
31          tmr.alarm(6, 1000, 1, send_ping)
32      end)
33  end
34
35  local function ds18b20_start()
36      print("ds18b20_start....\n ")
37      -- pin sensor
38      --gpio0 = 3 --D3
39      gpio3 = nil -- default D9
40      gpio4 = 2 --D4
41
42      ds18b20.setup(gpio3)
43      addrs = ds18b20.addrs()
44      if (addrs ~= nil) then
45          print("Total DS18B20 sensors: "..table.getn(addrs))
46      end
47      -- just read temperature
48      print("Temperature: "..ds18b20.read().."'C\n")
49      print("Temperature: "..ds18b20.read(nil,ds18b20.K).."'K")
50      -- release it after use
51      ds18b20 = nil
52      ds18b20 = nil
53      package.loaded["ds18b20"]=nil
54  end
55
```

```
56 function module.start()
57   ds18b20_start()
58   --mqtt_start()
59 end
60
61 return module
```

E.4. config.lua

```
1  local module = {}
2
3  -- wifis array
4  module.SSID = {}
5  module.SSID["MOVISTAR_*"] = "pass"
6  module.SSID["MOVISTAR_*"] = "pass"
7  module.SSID["Orange_*"] = "pass"
8
9  module.HOST = "test.mosquitto.org"
10 module.PORT = 1883
11 -- topic
12 module.ENDPOINT = "nodemcu/"
13
14 return module
```